AltiVec

(a.k.a Velocity Engine)

by

Ian Ollmann, Ph.D.
iano@cco.caltech.edu

Abstract

This tutorial covers the concepts, instructions and tools that you will need to master in order to accelerate your program using the G4's vector unit. Why would you want to spend the effort? Architectural improvements enable the vector unit to outpace the integer unit by up to a factor of 16 and the FPU by up to a factor of four. In practical application, improvements in data layout calculation efficiency and cache usage can show speed increases many fold higher than that. It is all about speed. Well, almost... The vector unit is also just plain cool.

This work and accompanying sample code is Copyright © 2001-2003 by Ian Ollmann, the author. All Rights Reserved. This work may be reproduced in full or in part only after obtaining expressed written consent of the author. You may use the sample code found in this document or in the accompanying sample code or derivative works thereof in your own software without restriction. This material is presented for educational purposes only. No warranty as to its correctness or suitability or merchantability for any particular purpose is stated or implied. Use at your own risk.

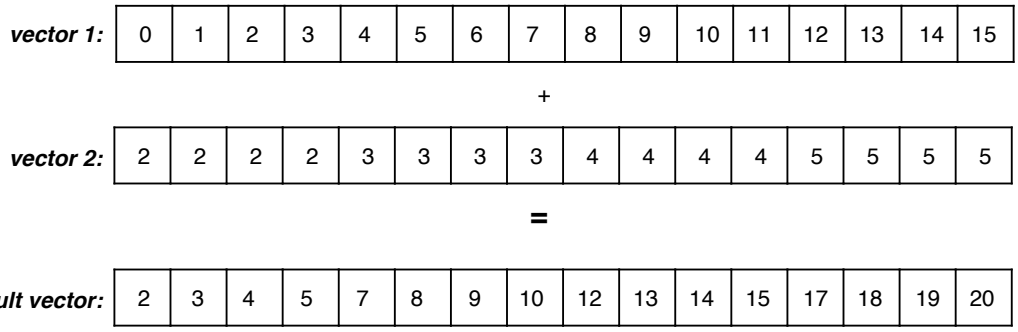
Introduction

The G4's AltiVec unit (vector unit) is a new computation unit added to PowerPC separate from the integer unit and FPU (the scalar units). The difference between the vector unit and the scalar units is that the vector unit handles multiple pieces of data simultaneously in parallel with a single instruction. This format is called SIMD (Single Instruction Multiple Data). Recent x86 processors also have a SIMD unit that does similar things. (MMX, SSE and SSE2)

To illustrate the difference between the scalar way of doing things and the vector way, we can look at a simple operation like addition. One might write 1 + 1= 2 in the integer unit, and 1.0 + 1.0 = 2.0 in the FPU. In the vector unit, one would write:

```
//Add vector1 to vector2 and place the result in resultVector
resultVector = vec_add(vector1, vector2).
```

What is the difference? Each 128-bit vector can hold up to 16 different numbers at once. As a result, the addition looks like this:



As you can see, each element from `vector1` is added to the corresponding element of vector2 and the result is stored in the `result vector`. Vector integer addition takes only a single cycle, the same amount of time as the scalar versions. Thus, it should be immediately clear why the AltiVec unit is so fast. By working on data in parallel, you get a lot done!

In the "Getting Started" section, we will begin this tutorial with a discussion of basic concepts behind the way that the vector unit operates. Then, this tutorial will introduce the different AltiVec data types and the operations that can be performed on them. We will continue with a discussion of factors that contribute to code performance or lack thereof. This tutorial will conclude with some practical wisdom to guide your work with AltiVec.

Getting Started

Invaluable Documentation

If I manage to do anything right in this tutorial, it should be to direct you to the **Motorola documentation** on the vector unit. There is both an assembly programming interface and a C programming interface, and they have different instruction manuals. Get both. Neither is really complete without the other:

http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/ALTIVEC/ALTIVECPEM.pdf
http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/ALTIVEC/ALTIVECPIM.pdf

Motorola updates these from time to time. It is worthwhile to check for newer revisions if your copy is starting to look a little dog-eared. In addition, you will find a number of other AltiVec resources enumerated at the end of this document in the section "**Other Resources**".

In addition, Apple Computer has updated their Velocity Engine site with extensive practical information, benchmarks, algorithms and programming tips, so it is worth a visit:

http://developer.apple.com/hardware/ve

Apple also provides a extensive vector library, called vecLib.framework. To see what is available, look at /System/Library/Frameworks on your mac, if you are using MacOS X, to see what is available. It has extensive DSP functionality, BLAS, LAPACK, support for multiprecision arithmetic and also a vector math library for functions like sin(), cos(), pow(), etc.

Programming Interfaces

You may program for AltiVec using either the C interface or the assembly (asm) interface. For the most part, the C interface is the same as the asm interface, except that you get to pretend you are using C functions instead of assembly instructions, and the compiler is able to schedule the instructions for you for optimum performance. The actual C "functions" largely map 1:1 with AltiVec instructions, so the vocabulary itself doesn't vary much between the two languages.

You can program in asm, C or C++ using **CodeWarrior**, and **MrC/MrCpp** on MacOS 9 and earlier. You can program in C, C++ or Obj C on MacOS X, using **CodeWarrior** and **GCC** (or **Project Builder**). GCC and Project builder come free with MacOS X. Some other languages and compilers also support for AltiVec (**Lightsoft Fantasm** for asm is an example) and **Absoft** for FORTRAN. In the rest of cases, it is often sufficient to write AltiVec libraries in C as a shared library and call them from other environments, such as **RealBasic**.

AltiVec does not work on any pre-G4 computer. If you issue an AltiVec instruction on a pre-G4 it will probably crash with an illegal instruction exception. Apple briefly flirted with some **emulation libraries** in MacOS 8.1 for PPC 603/604/750, but these are not suitable for deployment in final code. Any AltiVec emulator is likely going to be extremely slow. If you need to deploy on both pre-G4 and G4 you will be writing two versions of every function that uses AltiVec.

AltiVec Data Types

AltiVec vectors are all 16 bytes in size (128 bits). This is the size of a single AltiVec register. There are 32 AltiVec registers in the processor. The data in a vector register can be interpreted and processed as 128 bits, 16 chars, 8 shorts, 8 16-bit pixels, 4 ints, or 4 single precision IEEE-754 floats. The integer types may be signed or unsigned. 32-bit pixels can be interpreted either as 4 unsigned ints or 16 unsigned chars, depending on whether you want to manipulate them on a per pixel or per channel basis. The vector unit does not do double precision floating point. This is not a large loss, since at most you could expect 2:1 parallelism. To help soften the blow, Motorola refined the FPU so that all double precision operations now take the same amount of time as their single precision versions, with the exception of division. If you need to do double precision math, just use the FPU. There is probably quite a lot you can do to optimize your code there.

In the C interface, each of these new types is named by placing the **vector** keyword before the type of element in the vector.

8 bit types	16 bit types	32 bit types
vector char	vector short	vector int
vector unsigned char	vector unsigned short	vector unsigned int
	vector pixel	vector float

Each of these can be thought of as a small array of values stuffed into a single container:

8 bit types	16 bit types	32 bit types
char[16]	short[8]	int[4]
unsigned char[16]	unsigned short[8]	unsigned int[4]
	(arrrrrgggggbbbbb)[8]	float[4]

In fact, you can write a union in C to convert between the two notations. Here is an example that overlaps an array of 8 shorts in memory with a single vector short:

```
//Unions allow access to data as either individual elements
//or whole vectors
typedef union
{
       vector short        vec;
       short               elements[8];
}ShortVector;
```

This device is a particularly useful way to get member-wise access to a vector. Typically this is not easily accomplished in the vector unit. Instead, you may use the union as a clean way to shuttle the data from vector unit to FPU/IU by way of the stack. The union approach,

```
//An easy to read way to get the third element out of a vector
ShortVector shortVector;

shortVector.vec = (vector short) someVectorShort;
theThirdElement = shortVector.elements[2];
```

is much easier to read and more ANSI C compliant than a lot of typecasting and pointer dereferencing:

```
//Hard to read way to get the third element out of a vector
theThirdElement = *( ((short*) &someVectorShort )+ 2 );
```

They both compile to the same thing, so save yourself a few bugs and make use of unions.

AltiVec Operations

AltiVec comes with a series of different operations that you can perform on vectors. These are divided into several different classes: **type converters**, **constant initializers**, **mathematical operations**, **Boolean operations**, **comparators**, **memory operations** and **permute operations**. The various AltiVec instructions are all described in the **AltiVec Programmers Instructions Manual** in detail, sorted alphabetically. Here I present them arranged by function to help you find the right instruction for a given task.

Type Converters

The different vector types are interconvertable between types, without having to do any work as long as the bit pattern doesn't change. For example if the result of an instruction (e.g. `vec_splat_u8`) is a vector unsigned char, and you need a vector float with the same bit pattern, you can simply typecast the one to the other. Unlike conversions between classic floating point and integer types, this operation is free:

```
//Convert a vector unsigned char to a vector float
vector float zero = (vector float) vec_splat_u8(0);
```

It is important to remember that all such conversions do not change any bits. In this particular example `0x00000000` just happens to have the value `0.0` when interpreted as a float. If you did the same operation using 1 instead (giving `0x01010101`), the result would not be 1.0 when interpreted as a `vector float`.

To convert values between integer and floating point representations with retention of *numerical value*, use the appropriate format conversion function. These are `vec_ctf`, "vector convert to float", `vec_ctu`, "vector convert to unsigned fixed word saturated" and `vec_cts`, "vector convert to signed fixed point word saturated". Since these have no stack overhead associated with them, they are a lot faster than the scalar conversions that do the same thing. In some cases, the difference has been observed to be up to a factor of thirty. If you want a vector float full of 1.0, you may generate it from the corresponding integer version in this fashion:

```
//Generate a vector float constant holding {1.0, 1.0, 1.0, 1.0}
vector float one = vec_ctf( vec_splat_u32(1), 0 );
```

Conversions between different integer types can be done using `vec_pack()` or `vec_unpackh()` and `vec_unpackl()`. `Vec_packs()` and `vec_packsu()` do saturated conversions from larger types to smaller types. (`Vec_packsu()` is for unsigned types.) `Vec_packpx ()` does the 32 → 16-bit pixel conversion. `Vec_unpackh()` and `vec_unpackl()` are used to convert 16 bit pixels back to 32 bit pixels. Here is a simple chart to help you sort out what does what:

From\To:	char	short	int	16-bit pixel	32-bit pixel	float
char	-	vec_unpackh, vec_unpackl	Convert to short first	**X**	**X**	Convert to int first
short	vec_pack, vec_packs, vec_packsu	-	vec_unpackh, vec_unpackl	Static typecast	**X**	Convert to int first
int	Convert to short first	vec_pack, vec_packs, vec_packsu	-	**X**	Static Typecast	vec_ctf
16-bit pixel	**X**	Static typecast	**X**	-	vec_unpackh, vec_unpackl	Convert to 32-bit pixels first
32-bit pixel	**X**	**X**	Static typecast	vec_packpx	-	Convert channels to ints
float	Convert to ints first	Convert to ints first	vec_ctu, vec_cts	Convert to 32-bit pixel first	Convert to int first	-

Static Initializers

Introducing constants into the vector unit is a task that frequently needs to be done. The simplest way to do this is to just load in a constant from global storage:

```
//Load {0.0, 1.0, 2.0, 10.0} into myVector
vector float myVector = (vector float) ( 0.0, 1.0, 2.0, 10.0 );

//For the Red Hat / FSF compiler use curly braces instead
vector float myVector = (vector float) { 0.0, 1.0, 2.0, 10.0 };
```

Unfortunately, this is typically rather expensive. If the memory location where the constant lives doesn't happen to be in the cache, you might be hit by as much a 35–250 cycle memory stall. If your function is likely to take less than 200 cycles, then it is often significantly to your advantage to avoid loading in constants this way. It is also nicer to the caches.

Happily, there is a series of `vec_splat_X#()` functions available to help you generate constants. They can produce vectors filled with some value in the range –16…15, as either vector char, short or int. For example, `vec_splat_u8(1)` will produce a vector full of `0x01`, whereas `vec_splat_s32(1)` will produces a vector full of `0x00000001`. For some hints about how to create larger vector types, please visit Holger Bettag's site:

http://www.informatik.uni-bremen.de/~hobold/AltiVec.html

The other type of static initializer is the pair of instructions `vec_lvsl` and `vec_lvsr`. These produce vector unsigned chars that count upwards from one element to the next. By design, they are for quickly generating constants to deal with memory alignment. However they prove to be much more useful than that. Quite often you need vector constants which have different values stored in each element. These cannot be generated using the `vec_splat_XX()` functions. Surprisingly often, these can instead be generated starting from the vectors generated by `vec_lvsl` or `vec_lvsr`. In addition, **`vec_lvsl` and `vec_lvsr` are the only two vector instructions that allow you to move an integer from the integer unit to the vector unit directly without any load/store overhead.** If you need to transfer four bits from the integer unit to the vector unit quickly, these can do it in 2-3 cycles. If you do it through the stack instead, it probably will take 7-10 cycles for a full 32 bit quantity splat across a vector register. A full 128 bits probably takes at least twice as long.

Mathematical Operations

Addition and Subtraction

Addition and subtraction are handled through `vec_add()` and `vec_sub()`. If you wish to avoid integer overflow, there are special variants, `vec_adds()` and `vec_subs()` that do saturated addition or subtraction between corresponding elements of two vectors. These are particularly handy for pixel transparency operations for example, where overflow will cause highly visible dark pixels in what should be a bright area. Another use might be in sound, where

an overflow clipped improperly would cause a crack or pop. In addition, there are also `vec_sum4s()` and `vec_sum2s()` that add across single vectors for specific integer types.

Multiplication

There are quite a number of different flavors of multiplication: `vec_madd()`, `vec_madds()`, `vec_mladd()`, `vec_mradds()`, `vec_msum()`, `vec_msums()`, `vec_mule()`, `vec_mulo()`, and `vec_nmsub()`. These are typically specialized by vector type — most operations only apply to particular data types. For the most part, they actually perform a multiply-addition operation (result = A * B + C) rather than a simple multiplication. If you just need multiplication, pass a vector full of zeros for C. For floating point math, it is advantageous to pass a vector full of negative zeros instead. Please see the following section about doing full precision reciprocal square roots for how to quickly prepare the negative zero constant.

Division

Division is only directly possible using vector floats. This is done using `vec_re()`. `Vec_re()` does a half precision estimate of the reciprocal of a value. The full precision reciprocal is obtained by doing a single Newton-Raphson refinement step:

```
//Reciprocal with Newton Raphson refinement
inline vector float vec_reciprocal( vector float v )
{
	vector float reciprocal = vec_re( v );
	return vec_madd( reciprocal,
			 vec_nmsub( reciprocal, v, vec_float_one()),
			 reciprocal );
}

//Generates (vector float)(1.0)
inline vector float vec_float_one( void )
{
	return vec_ctf( vec_splat_u32(1), 0);
}
```

Integer division can be accomplished using `vec_mradds()` to do an integer multiply by a fixed point reciprocal. The instruction essentially does this operation:

$$\textbf{Saturate\_to\_SInt16(}\ \frac{A \square B + 16384}{32768} + C\ \textbf{)}$$

If *B* is the *signed* 0.15 fixed point representation of the reciprocal of your divisor and *C* is zero, then the operation amounts to full precision *signed* 16 bit division. The below function will calculate signed 16 bit division to 12 bits of precision. If you need a full sixteen bits of precision, then you will have to substitute the `vec_reciprocal()` code above for where I have written `vec_re()`.

```
//Signed 16 bit integer division. Note: accurate to only 12 bits. If more is
//required. use Newton-Raphson
vector signed short Divide16(  vector signed short numerator,
                               vector signed short denominator)
{
    vector signed short zero = vec_splat_s16(0);

    //Convert denomenator to 1/denomenator as a signed 0.15 fixed point
    vector signed int denom1 = vec_unpackh( denominator );
    vector signed int denom2 = vec_unpackl( denominator );
    vector float recip1 = vec_re( vec_ctf( denom1, 0 ) );
    vector float recip2 = vec_re( vec_ctf( denom2, 0 ) );
    denom1 = vec_cts( recip1, 15 );
    denom2 = vec_cts( recip2, 15 );
    denomenator = vec_pack( denom1, denom2 );

    //return numerator * 1/denominator
    return vec_mradds( numerator, denominator, zero );
}
```

Because of pipelining issues in the VFPU, this would be faster if it handled division of sets of two vector shorts simultaneously. `Vec_reciprocal()` would be faster if it handled four vector floats.

Square Roots and Reciprocal Square Roots

Square roots are also only possible with floats. The `vec_rsqrte()`, "vector reciprocal square root estimate" instructions returns a half precision result. A further Newton-Raphson refinement step is required for a full precision reciprocal square root:

```
//Calculate the full precision reciprocal square root of v
inline vector float vec_reciprocal_sqrt( vector float v )
{
       const vector float kMinusZero = vec_neg_zero();
       const vector float kOne = vec_ctf( vec_splat_u32( 1 ), 0 );
       const vector float kOneHalf = vec_ctf( vec_splat_u32( 1 ), 1 );

       //Calculate 1/denomenator using newton rapheson
       //refined reciprocal estimate
       vector float sqrtReciprocalEstimate = vec_rsqrte( v );
       vector float reciprocalEstimate = vec_madd( sqrtReciprocalEstimate,
                                                   sqrtReciprocalEstimate,
                                                   kMinusZero );
       vector float halfSqrtReciprocalEst =
           vec_madd( sqrtReciprocalEstimate, kOneHalf, kMinusZero );
       vector float term1 = vec_nmsub( v, reciprocalEstimate, kOne );

       return vec_madd( term1, halfSqrtReciprocalEst,
              sqrtReciprocalEstimate );
}

//Generate a vector full of -0.0.
inline vector float vec_neg_zero( void )
{
       vector unsigned  result = vec_splat_u32(-1);
       return (vector float ) vec_sl( result, result );
}
```

To generate a square root from a reciprocal square root, simply multiply the original value by the reciprocal square root: $x^{1/2} = x * (1 / x^{1/2})$.

> Why is –0.0 used in the above example? The circuitry in the vec_madd function (vmaddfp) only gives IEEE-754 corrrect answers for edge cases if –0.0 is passed instead of 0.0. For most code, the difference may not be important.

Miscellaneous Floating Point Operations

There is an estimator for exponentials, `vec_expte()` which estimates the value of 2^{value}. Likewise, there is a an estimator for the base-2 logarithm of a floating point quantity, `vec_loge()`. `Vec_ceil()` and `vec_floor()` perform the same operation as the analogously named C Std Library functions.

Miscellaneous Integer Operations

`Vec_avg()` takes the average of two integers (rounding upwards). `Vec_abs()` and `vec_abss()` may be used to take the absolute value of integers. They are one of the few C functions available that do not directly map to a single AltiVec instruction.

Boolean Operations

All of the expected Boolean operations can be done on entire vectors via `vec_and()`, `vec_or()`, and `vec_xor()`. In addition, there are the and with complement (`vec_andc`, A &~B) and NOR operators (`vec_nor`, ~(A | B)).

Comparators

There are a lot of ways to determine how two different vectors relate to one another. There is a series of `vec_cmpXX()` instructions that determine element-wise relationships between two vectors such as less than, greater than or equal, equal, etc. There is a `vec_cmpb()` function that checks which elements fall between the bounds defined by two other vectors. The results from these functions are **–1** for all elements where the test is true. These work nicely with `vec_sel()` for various tasks.

There is a series of comparators that store their results in a register in the integer unit. These come in two flavors. There is a series of `vec_any_xx()` functions that return **1** if any element satisfies the test. There is also a series of `vec_all_xx()` functions that return **1** if all of the elements in the vector satisfy the test. These return 0 if not true. The tests are not limited to inequalities. You can also test for NaN (not a number), whether or not a floating-point quantity is numeric, or out of bounds.

You can do elementwise compares of vector floats by taking advantage of the behavior of `NaN`. Any compare against `NaN` returns `false`. So, for example, if you want to know if the first element of a vector float is greater than zero, you can use `vec_any_gt()`:

```
// NaN in IEEE-754 float
#define QNaN      0x7fc00000UL        //Quiet
#define SNaN      0x7f800000UL        //Throws an exception (signalling)

//Return true if the first float in v is greater than 0.0
Boolean IsFirstElementPositive( vector float v )
{
    vector unsigned int compare = (vector unsigned int)( 0, QNaN, QNaN, QNaN);
    return vec_any_gt( v, (vector float) compare );
}
```

The `vector unsigned int` to `vector float` type conversion is required because `QNaN` is not a standard C quantity. It is loaded in as a unsigned int to give us bitwise control over input value. `0xFFFFFFFFUL` is also a `QNaN`. Thus the compare constant also could have been synthesized on the fly this way without touching memory:

```
vector signed char compare = vec_sld( vec_splat_s8(0), vec_splat_s8(-1), 12 );
```

The negated compare functions (`vec_aXX_nXX()`) reverse this trend — where `NaN` appears, the result is always `true`. So if you want to know if the second and third elements are greater than zero, you can use `vec_all_nle()`:

```
//Return true if the second and third floats in v are greater than 0.0
Boolean AreSecondAndThirdElementsPositive( vector float v )
{
    vector unsigned int compare = (vector unsigned int)( QNaN, 0, 0, QNaN);
    return vec_all_nle( v, (vector float) compare );
}
```

Perhaps you were wondering why the negated compares were there!

Finally, the other half of the compare is responding to the results. What do you do with the mask full of 0's and 1's that you get from a `vec_cmp*` function? Most of the time, what you do is calculate both sides of the branch and then use `vec_sel()` with the mask to select the correct results. You evaluate both sides of the branch because some parts of the vector might have gone down one path and parts of the vector might go down the other. Here is an example. Let's vectorize a MAX function:[1]

[1] Note that we already have vec_max() for this purpose. This is just an example.

```
//Return the maximum of two integers
int Max( int a, int b )
{
	int result;

	if( a < b )
		result = b;
	else
		result = a;

	return result;
}
```

This is written in vector code as:

```
//Return the maximum of two vector integers
vector signed int Max(vector signed int a, vector signed int b )
{
	vector bool int mask = vec_cmplt( a, b );	//If ( a < b)...
	vector signed int result = vec_sel( a, b, mask ); //Select a or b

	return result;
}
```

Notice that there is no branching here. This is very good for speed since there is no possibility of taking a multicycle branch miss. `Vec_sel()` can be thought of as doing the following logical operation:

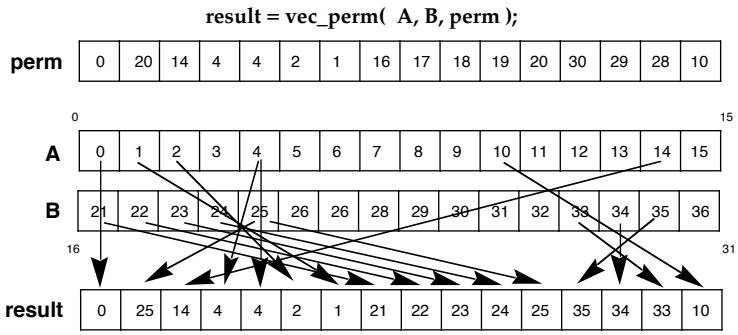
```
result = (a & ~mask) | (b & mask);
```

Permute Operations

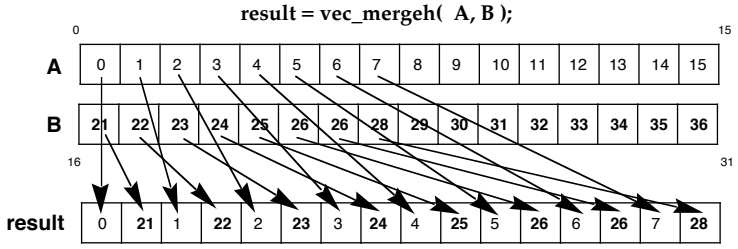
It is quite often the case that you need to swap elements from within a vector or among two vectors.

> ▲ CAUTION: If you are using a lot of permutes, your code may have poor parallelism. Consider other algorithms and data arrangements requiring less data reorganization. Obviously some code exists merely to move data around or reorganize it. This caution mostly applies for those functions for which a scalar versions would not be doing a lot of data copying. ▲

The most generic of the permute functions is `vec_perm()`. It can take any arbitrary collection of bytes from two source vectors and shuffle them into a third vector. To do so, it uses a permute vector that holds values between 0 and 31 to indicate which byte from the two parent vectors to use for each cell. While vec_perm() is quite efficient itself, it is often somewhat costly to create the permute vectors for it to use. For that reason there are some other permute operations that do fixed swap types for common operations.



`Vec_mergeh()` and `vec_mergel()` are two such operations. They may be used to interleave the contents of two vectors with one another, much like shuffling a deck of cards. This is very useful for expanding unsigned integers to the next larger size, by merging with a vector full of zeros. They are also a very efficient way of doing matrix transposes.



You may use a number of different vector operations for doing left and right shift operations. For full 128 bit shifts, you use a combination of `vec_slo()` and `vec_sll` (left shifting) or `vec_sro` and `vec_srl` (right shifting).

> ▲ CAUTION: Be sure to read the instructions carefully for these. The shift bitcount is stored in the rightmost element of the shift vector, but **the results are undefined** if all bytes in the vector do not contain the same value. Careful note should also be taken of which bits on each byte are used to determine the shift amount as they are different for the shift and shift by octet versions. This is so a full 128 bit shift can be performed by successive calls to `vec_slo()` and `vec_sll()` or `vec_sro()` and `vec_srl()`, using the same shift constant for each. ▲

```
//Shift a vector left by the number of bits indicated
vector unsigned char vec_shift_left( vector unsigned char v,  Uint8 bitCount )
{

   //Load the bit count into one of the bytes in our vector
     vector unsigned char shiftValue = vec_lde( 0, &bitCount );
     vector unsigned char splatByteAcrossVector;

     //Load the bit count into the all elements of a vector
     splatByteAcrossVector = vec_splat( vec_lvsl( 0, &bitCount ), 0 );
     s = vec_perm( shiftValue, shiftValue, splatByteAcrossVector );

     //Shift left
     return vec_sll( vec_slo( v, s), s);
}
```

There are some element-wise shift instructions that rotate or shift each element by a number of bits corresponding to the value held in an analogous position in another vector. These are `vec_sl()`, `vec_sra()` and `vec_sr()`. Finally, there is `vec_sld()` which may be used to rotate or shift left vectors by a fixed number of bytes. This is especially useful for concatenating vectors and for rotating a vector by a known amount. (See the sample code for **Dot Product** below.)

Memory Operations

Loads and Stores

If you are using the *AltiVec* C extension and are writing in C/C++/Obj C, you can use standard C notation to perform loads and stores as you would for scalar quantities, except that the vector alignment rules apply (the address is silently rounded down to a multiple of `sizeof(vector)`):

```
//Load some data from memory using standard C notation
vector float *dataPtr = someAddress;
vector float theData = *dataPtr;      //rounded silently as (dataPtr & ~15)
vector float moreData = dataPtr[1]; //rounded to ((dataPtr+16 bytes) & ~15)
```

However, this is not flexible enough to accommodate all the different ways to load and store data from vector registers. For this reason, the AltiVec C extension defines a set of functions for explicitly loading from and storing to memory. The simplest are `vec_ld()` and `vec_st()`. These simply load and store single vectors much like the example above. The loads and stores align themselves to the nearest 16 byte aligned block — the address you specify is internally truncated to a multiple of 16 in hardware before any loads or stores are performed. If the vector you seek is not aligned, you will need to handle the alignment yourself in software. `Vec_lvsl()` or `vec_lvsr()` may be used to generate the permute vector that you will need to generate the unaligned vector from two adjacent aligned vectors:

```
//Load a vector from an unaligned location in memory
vector unsigned char LoadUnaligned( vector unsigned char *v )
{
    vector unsigned char permuteVector = vec_lvsl( 0, (int*) v  );
    vector unsigned char low = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v );
    return vec_perm( low, high, permuteVector );
}

//Store a vector to an unaligned location in memory
void StoreUnaligned( vector unsigned char v, vector unsigned char *where)
{
  //Load the surrounding area
    vector unsigned char low = vec_ld( 0, where );
    vector unsigned char high = vec_ld( 16, where );

  //Prepare the constants that we need
    vector unsigned char permuteVector = vec_lvsr( 0, (int*) where );
```

```
    vector signed char oxFF = vec_splat_s8( -1 );
    vector signed char ox00 = vec_splat_s8( 0 );

  //Make a mask for which parts of the vectors to swap out
    vector unsigned char mask = vec_perm( ox00, oxFF, permuteVector );

  //Right rotate our input data
    v = vec_perm( v, v, permuteVector );

  //Insert our data into the low and high vectors
    low = vec_sel( v, low, mask );
    high = vec_sel( high, v, mask );


  //Store the two aligned result vectors
    vec_st( low, 0, where );
    vec_st( high, 16, where );
}
```

`StoreUnaligned()` probably looks like a lot of work to you. *It is*. You should do your utmost to make sure that data is aligned properly. If you can't, then often you can fold much of this overhead into a loop when working with large arrays in a way that you only really pay for the alignment at the edges of the array.

> ▲ CAUTION: When handling unaligned loads and stores, it is important to make sure that you do not load or store data to memory locations that you do not know exist. If your load or store wanders off a existing page onto an unmapped one, your program may terminate with an unmapped memory exception or segmentation fault. This is a common danger for code that reads in an unaligned vector that turns out to be 16 byte aligned. In such cases, the second aligned load will likely contain no needed data and may be to indeterminate memory. ▲

In some cases, it may be appropriate to use `vec_ldl()` and `vec_stl()` instead of `vec_ld()` and `vec_st()`. These load and store in a manner that is more friendly to data already in the caches. `Vec_ldl()` and `vec_stl()` mark new cache blocks as those least recently used. This means that they will be the first to be flushed when more space in the cache is needed. Furthermore, `vec_ldl()` and `vec_stl()` mark their cache blocks as transient, which means that they will be flushed directly to memory rather than take up space in the L2. If you don't mark a cache block transient, it is flushed to the L2 cache, which serves as a victim cache for the processor. This may cause data that you need in the L2 cache to be flushed to RAM to make room. (The only way to get data into the L2 on the G4 is to displace it out of the L1.) If you are likely to have any data that you need to stay in the L2, using `vec_ldl()` and `vec_stl()` with other data that doesn't need to be in the L2 can be a good way to make sure your important data is not flushed to RAM.

Be warned that `vec_ldl()` and `vec_stl()` typically appear to slow your function down in benchmarking routines. This is because they cause more direct stores to memory that bypass the L2 cache. This slowdown is meant to be offset by an overcompensating speed gain in functions called around the function using `vec_ldl()` or `vec_stl()`. However in benchmark programs

these typically are not there. Thus, the positive performance impact for `vec_ldl()` and `vec_stl()` can usually only be correctly measured in your full program.

There are single element versions for load and store called `vec_lde()` and `vec_ste()`. These operate on one element at a time rather than a whole vector at a time. Which element is stored or loaded to depends on the alignment of the target. These variants on `vec_ld()` and `vec_st()` are fairly inefficient to use, but do have their occasional uses. One example would be after a vector dot product, to move the result to the FPU:

```
//Note that it is often more efficient to do dot products of small
//vectors in the FPU.
inline float DotProduct( const FVec v1, const FVec v2 )
{
   float result;
   vector unsigned  temp = vec_splat_u32(-1);
   vector float minusZero = (vector float ) vec_sl( temp, temp );

   // find the dot product of the two vectors
   vector float length2 = vec_madd( v1, v2, minusZero );

   //Sum across all elements
   length2 = vec_add(length2, vec_sld(length2, length2, 4 ) );
   length2 = vec_add(length2, vec_sld(length2, length2, 8 ) );

   //All elements in length2 are now the same. Store the result to
   //the stack and load it in to the FPU and return it in an FPU
   //register
   vec_ste( length2, 0, &result );
   return result;
}
```

> Note: I have seen a few code examples in which the developer used exclusively single element vector loads and stores as a crutch to avoid having to handle data alignment himself. While he was successful in avoiding the need to deal with memory alignment, this approach is not recommended for performance reasons. The LSU is enough of a bottleneck without quadrupling (at a minimum) the amount of work it has to do. This approach will also not correctly handle the case in which the elements themselves are not aligned to their natural alignment. Vector element load and store addresses are truncated in hardware to 2 or 4 byte aligned addresses in a fashion similar to how full vector loads are truncated to 16 byte aligned addresses.

Streaming Cache Instructions

Some of the most important additions to AltiVec are the streaming cache instructions. Quite a lot of what you will be doing in AltiVec will be rate limited by memory overhead, which is to say that the speed of the memory bus and not the CPU or your code will be determining the speed of your function. The streaming cache instructions may be used to directly manipulate how and when data is used in the caches. Typically these are more efficient than their older cache hint bretheren such as `dcbt`. This is because single streaming cache instruction can be used to fetch a lot of data. In addition, you have a little bit more control over what happens to the data after you are done with it.

You may set up from one to four cache streams using `vec_dst` (load) or `vec_dstst` (load + store). There are "transient" variants called `vec_dstt()` and `vec_dststt()` that mark their blocks to be flushed straight to RAM when they become stale instead of being sent to the L2 and L3 caches. This is a tool to help you preserve other valuable data already in the L2 cache that may be needed later. Use `vec_dst()` and `vec_dstt()` when you intend to just read a piece of data. Use `vec_dstst()` and `vec_dststt()` when you plan to read and modify a piece of data. *If you tend to only write a piece of data, you should not prefetch the data.* The G4 put all such writes into a store miss merge queue which will handle the cache details for you. When making two adjacent vector stores to the same cacheline, there are significant performance advantages to placing the stores one after the other in the instruction stream. This allows the hardware to avoid having to load the data from memory only to overwrite it. In such situations, use of `dcbz` is not productive.

> Note: On the PowerPC 7450 and 7455 (G4 Macintoshes above about 600 MHz), there is some advantage to doing loads out of order in the event of a L1 cache miss. Please see section 6.7.6.5.5 of the **MPC7450 RISC Microprocessor User's Manual** for more details.

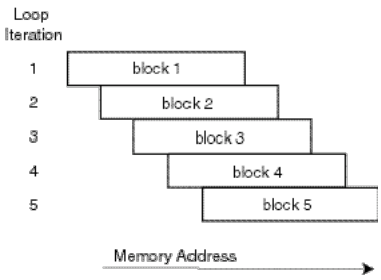
All of these streaming cache instructions are configured using a data stream prefetch control constant that sets the size of each block in vectors, how many blocks there are and the distance between them in bytes. For detailed information about how to configure these, please see the **AltiVec Programming Environment manual**, page 5-3. (It's not in the **Programming Interface Manual**.) As a bit of a shortcut, this function will make the process a bit easier until you get used to it. Note that there are some strict limitations as to the size of the blocks, so be sure to become familiar with these or you may end up getting something other than what you thought you would get:

```
//Initialize a prefetch constant for use with vec_dst(), vec_dstt(), vec_dstst
//or vec_dststt
inline UInt32 GetPrefetchConstant(    int blockSizeInVectors,
                                      int blockCount,
                                      int blockStride )
{
    ASSERT( blockSizeInVectors > 0 && blockSizeInVectors <= 32 );
    ASSERT( blockCount > 0 && blockCount <= 256 );
    ASSERT( blockStride > MIN_SHRT && blockStride <= MAX_SHRT );
    return        ((blockSizeInVectors << 24) & 0x1F000000)      |
                  ((blockCount << 16) && 0x00FF0000)             |
                  (blockStride & 0xFFFF);
}
```

The `blockSizeInVectors` must be a number between 1 and 32 (16 – 512 bytes). The `blockCount` must be 1 – 256. `BlockStride` is in bytes and should be in the range: -32768 – 32767. You may use negative strides to prefetch in the backwards direction.

There are four data streams available for use numbered from 0 to 3. If another thread or interrupt level process starts a new stream using the same number as yours then yours will be

stopped. (`BlockMoveData()` is an example of an interrupt safe MacOS function that uses stream 3. It is frequently called in the **Sound Manager** at interrupt level 4.) For this reason, the convention under the older MacOS (before MacOS X) is to count upwards from 0 from `SystemTask` level code, and count down from 2 in interrupt level code. Leave stream 3 to `BlockMoveData()`, unless you are writing code for a sound callback function, in which case use stream 3. Prefetch streams will also stop silently (without causing trouble) if they step on memory that is either unmapped or would cause a protection violation. (This includes memory currently paged out to disk as part of the operating system's Virtual Memory implementation.) On MacOS X, the data streams may stop for a wider variety of reasons, including simply losing the CPU to another thread, so it is less important which stream you use and more important that you restart it frequently.



http://developer.apple.com/hardware/ve/caches.html

Using Streaming Cache Instructions

Because the stream can be silently terminated for a wide variety of reasons, it is usually a bad idea to prefetch all of your data at once using a single call to vec_dst*(). You should instead prefetch your data in small overlapping blocks, perhaps 64–512 bytes in size. Up to 80-90% overlap between blocks is not uncommon, though it may be slightly more efficient to overlap less. Each block should start from the first byte of data that you intend to read immediately, and extend some number of bytes to cover data that you intend to use in the near future. How many bytes are best to prefetch can be difficult to predict, and typically must be determined by experimentation. Typically, one issues a prefetch instruction at the start of each loop iteration to cover data needed for the current loop iteration and perhaps the next 2-4 iterations after that. As the loop iterates, the region to be prefetched will march along at the pace of the calculation:

As each new prefetch is issued on the same stream, the old one is stopped and the new one takes its place and begins to work on cachlelines in the new block that are not already in the L1 cache. For this reason, it may be anticipated that there is little or no cost to large amounts of overlap between successive prefetch blocks. This can be shown experimentally to be the case. Requesting more data each time than we need immediately has the effect of initiating loads of needed data farther ahead of time.

One advantage of using streams to prefetch data is that if you manage prefetch the data into the caches before it is needed, the data prefetch and your function operate in parallel. This reduces the memory overhead of your function to near zero. Keep in mind however that if your code is very fast it will rapidly catch up with the data stream and begin to stall. (The memory bus can only transfer data so fast. Prefetching gives the bus a head start but doesn't make it any faster.)

Thus, this strategy is only 100% effective if your function is not processing data faster than it can be loaded. If your calculation is faster than the stream, your code will catch up to the head of the stream, stall for a while waiting for data, catch up to the head of the stream, stall again, catch up to the stream and stall some more. While it is busy stalling, the processor is doing nothing, just wasting cycles. These cycles can be reclaimed by finding something additional to do with the data in register while you wait for more to be loaded in from memory. Thus, in many cases you may accelerate your application by grouping a lot of smaller functions together into a few big ones such that you can do more work per load/store pair. You may also choose to use this extra time to do an more expensive, more correct calculation.

> Why does the processor do nothing while stalled for a load? It doesn't exactly do nothing. It can have up to 8 instructions in flight (16 on a PPC 745x), so the next 7 (or 15) instructions after the load can continue to execute, as long as they didn't need the data from the load. However, those 7 (or 15) instructions can't retire, so they will stall in the execution units' pipelines and eventually fill those up too. You will usually see this in **Sim_G4** as a "CB Full" (Completion Buffer Full) stall, with a nearby load taking many (potentially dozens) of cycles to finish executing. Be aware that the default **Sim_G4** memory model is for the 60X bus, which Apple only shipped on some very early entry level G4's, so most such stalls are hints that stalls may be there, not necessarily actual stalls. You can detect actual stalls of this type using the CPU performance monitor registers. Shikari (part of Apple's **CHUD SDK**) is a tool that will help you do this easily.

When you are finished with a data stream, call `vec_dss()` to stop it. (Typically this is done at the end of your function once the main blitter loop has exited.) If you prefer, you may call `vec_dssall()` to stop all streams. Typically the stream is set up to read a few more bytes than you actually need due to the multiple overlapping calls to `vec_dst()` and variants. For this reason, it makes sense to stop it as soon as possible once you are done with the stream to prevent any more memory overhead than is strictly necessary.

Hardware Features that Affect Speed

It is necessary to have some understanding of the pieces that make up the G4 to get the most out of it. For complete documentation on the implementation of the PPC 7400, 7410, 7450 and 7455, processors please see their respective user manuals.

http://e-www.motorola.com/brdata/PDFDB/docs/MPC7400UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7410UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7410UMAD.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7450UMAD.pdf

Motorola also published a guide to help explain the differences between these processors.

http://e-www.motorola.com/brdata/PDFDB/docs/AN2203.pdf

Preliminary information about IBM's recently announced PowerPC 970 is available here:

http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/A1387A29AC1C2AE087256C5200611780

The Instruction Cache

When you enter a function, the instructions that make up that function have to be loaded in from memory. The G4 has a 32 kB 8-way set associative instruction cache to store them. However, it is impossible for the whole program to fit in the instruction cache unless it is a trivially small program. Thus, it is not unusual that a function may have to be loaded in from main memory, which can be very, very slow. This means that often the first loop in any function may be executed considerably more slowly than the successive iterations. This will show up in **Sim_G4** as large gaps when no instructions were operating. Typically these stalls can be from 35-40 cycles per set of eight instructions. (Eight PPC instructions fill up one cache block.)

Knowing this, it may be worthwhile to position blocks of code that are frequently called near one another close to one another in memory. In addition, knowing that you are likely to lose some time the first time through a function, it may be a good idea to start any data stream pre-fetch operations as soon as possible in the function. This will allow more data to be pre-loaded by the time the function starts operating at peak efficiency. Sometimes prefetching larger blocks at the start may help as well.

In addition, it may be generally true that it doesn't much matter how you write rarely called code as long as the instruction count is low. You have an average latency of four or five cycles per instruction, which is more than enough for almost everything. This is one more reason to only optimize the small part of the program that consumes 80% of the CPU's attention.

Pipelining

The number of cycles each instruction takes is listed in Table 6-9 of the **7400, 7410 and 7450 Users Manuals** (Chapter 6). For the most part, they take from 1-5 cycles each, depending on the vector subunit that they execute in. Most things take one cycle, except for operations in the **Vector Complex Integer Unit (VCIU)** and the **Vector Floating Point Unit (VFPU)**. The **vector permute unit** (VPERM) takes one cycle on PPC 7400 and 7410 but two cycles on the PPC 7450. The VCIU has three stages (four on PPC 7450) and the VFPU has four or five stages depending on whether or not Java mode is turned on. (Though the word Java is used here, this mode has little to do with the Java development platform. It is so named because it shares some numerical standards with the Java subset of IEEE-754.) Both the VCIU and the VFPU are pipelined, which means that you can have multiple instructions executing at once in each. Each cycle, one new instruction can be added to the pipeline and another one can exit out the other end and be retired. This

allows for a throughput of one instruction per cycle, even though it may take several cycles to process each instruction.

	PPC 7400 and 7410	PPC 7450 and 7455
VPERM	1	2
VSIU	1	1
VCIU	3	4
VFPU (Java mode)	4 (5)	4 (5)
Load from L1/L2/L3	2 / 11 / -	3 / 9 / 33

In order to make full use of the pipeline, you have to make sure that you have enough independent data available. Otherwise you will find that instructions will be prevented from starting down the pipeline in a timely fashion because they are waiting on the result of another calculation. As an example, suppose you are doing a vector dot product on a pair of very long vectors. A simple version might look like this:

```
//Simple dot product function for long vectors
float SlowDotProduct( vector float *v1, vector float *v2, int length )
{
    vector float temp = (vector float) vec_splat_s8(0);
    float result;

    //Loop over the length of the vectors multiplying like terms and summing
    for( int i = 0; i < length; i++)
          temp = vec_madd( v1[i], v2[i], temp);

    //Add across the vector
    temp = vec_add( temp, vec_sld( temp, temp, 4 ));
    temp = vec_add( result, vec_sld( temp, temp, 8 ));

    vec_ste( temp, 0, &result );
    return result;
}
```

The problem with this function is that each call to `vec_madd` depends on the result of the last one, so we don't actually get any pipelining here. We only do one `vec_madd()` every four or five cycles. A faster method would be to load 64 bytes from each vector each loop iteration. This would allow you to stuff the pipeline:

```
//Do v1 dot v2 faster. In this one we make sure the pipeline is full
float FasterDotProduct( vector float *v1, vector float *v2, int length )
{
    vector float temp = (vector float) vec_splat_s8(0);
    vector float temp2 = temp;
    vector float temp3 = temp;
    vector float temp4 = temp;
    vector float result;
```

```
    //Loop over the length of the vectors, this time doing
    //4 vectors in parallel to stuff the pipeline
    for( int i = 0; i < length; i += 4)
    {
           temp = vec_madd( v1[i], v2[i], temp);
           temp2 = vec_madd( v1[i+1], v2[i+1], temp2);
           temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
           temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
    }

    //Sum our temp vectors
    temp = vec_add( temp, temp2 );
    temp3 = vec_add( temp3, temp4 );
    temp = vec_add( temp, temp3 );

    //Add across the vector
    temp = vec_add( temp, vec_sld( temp, temp, 4 ));
    temp = vec_add( result, vec_sld( temp, temp, 8 ));


    //Copy the result to the stack so we can return it via the FPU
    vec_ste( temp, 0, &result );
    return result;
}
```

Clearly more can be done with this function, such as correctly handling the case when the vectors are not an even multiple of 64 bytes long. Also, some streaming cache instructions would really speed it up, since it is likely that a bigger bottleneck is memory overhead. However, it should benchmark a bit faster. You may examine some of the code examples that accompanies this tutorial to see similar examples about how ensuring proper feeding of the vector pipeline can bring more speed to your function.

The Data Cache

Like the instruction cache, the processor has several data caches for storing frequently used data. All data loaded into the processor causes the 32 byte cache block (a.k.a. cache line) associated with that piece of data to be loaded into the L1 cache. The L1 is a eight-way set-associative cache, 32 kB in size. What this means that each piece of addressable memory maps directly to a set of eight cachelines. (Different pieces of memory may map to different sets of 8). There are 128 such sets. When a 32 byte block is loaded into the cache, bits 20 to 26 of the address are consulted to determine which set to put the data in. Among the eight cachelines in the set, the one that is overwritten is chosen by a pseudo Least Recently Used (LRU) algorithm.

This organization can cause trouble in a rare number of cases. Since data spaced 4096 bytes apart maps to the same set, if you skip through memory reading data every 4096 bytes (or multiple thereof), you will end up using less than 1% of the cache because those data all map to the same set of cache lines and will quickly displace each other. As a general rule of thumb skipping

through memory with a stride that is a large power of 2 will make it likely that the data that you read in will be soon displaced. Very little of it will stay in the caches.

> Example: 1024 pixel wide GWorlds may have this problem if no padding is added to the right edge. The pixel rows are perfectly aligned with the organization of the caches. This means that each pixel row from a sprite as it is rendered into the drawing surface will compete with those before it for space in the cache.

The L1 cache is extremely fast. Most reads from it take only 2 cycles (3 on PPC 7450). By comparison, reads from main ram can take up to 250 cycles, though 30-50 is a more common number, depending on the number of cache misses, TLB misses, etc. Thus, proper use of the L1 is extremely important.

Typically one of the eight blocks in a set kept in the L1 is the top of the stack, due to the frequency with which it is accessed. It is possible that another one might be global storage. **Sim_G4** will quickly reveal any memory-related stalls that may be occurring due to poor use of the caches. You will see long stalls in `lvx` and `lvxl`.

When data is flushed out of the L1, it ends up in the L2. This is the only way for data to end up in the L2 cache. The L2 is called a 'victim cache' for this reason. The L2 cache is much larger than the L1, but is only 2 way set associative (8 way on the PPC 7450). If you have a particularly large set of data that you would like to stay in the L2, then you need consider using the transient cache instructions or the LRU load and store functions. These will prevent displaced data from being written to the L2, preserving data that is already there. Data access times to the L2 are still quite quick. They are 9 cycles on the 7450 and 10-15 cycles on the 7400 and 7410.

Another facet of memory management that can cause occasional problems is the TLB (translation lookaside buffer). Where memory is actually living in hardware is fairly complicated. Thus even though you have an address for it (e.g. 0xAC7E3500), actually locating it in hardware is a bit of work. Most memory is grouped together into pages, which are typically about 4 kB in size on MacOS. These can be stored in any of a number of places. The TLB caches 128 of these translated addresses (2 way set associative on 7400). If you need a piece of data from a page that is not in the TLB, then a rather expensive process of looking it up in a page table ensues. This can be more expensive than a cache miss. Based on 4096 bytes per page and 128 entries in the TLB, only about 512 kB of data can be referenced by the TLB.

Thus to make a long story short, for a number of reasons it is a very good idea to place data that is used together near one another in memory. That way they chance that they will share the same page or even the same cacheline are very high, and your code will not encounter many long memory associated stalls.

Alignment and Data Layout

AltiVec does not include hardware support for loading and storing unaligned vectors. A review of what is required to align vectors in software should quickly convince you that dealing with misaligned vectors can be slow and is almost always very, very complicated.[2] (Please see "**Memory Operations:** ***Loads and Stores***" above for a taste.) This is especially true of vector store operations (`vec_st` and `vec_stl`). If you must decide between unaligned loads and unaligned stores, pick unaligned loads. Unaligned stores would overwrite data adjacent to the target. Extra overhead is required to avoid this problem, and where thread safety is required, stores at the ends of the array must be done using vector element stores using `vec_ste()`.

The best possible solution is to simply align your data properly. MacOS heap blocks returned to you using `NewPtr()` or `OTAllocMem()` are already 16 byte aligned. Blocks from the MP heap allocated using `MPAllocateAligned()` can be aligned to suit your taste. On MacOS X, malloc() and its bretheren similarly return 16 byte aligned blocks. (For large allocations they are likely to even be page aligned.) Likewise, global and static storage starts 16 byte aligned at the start of every compilation block. Thus, all you have to do in general is make sure that your data and structs are properly arranged to preserve the 16 byte alignment that you are given. Vector types placed on the stack are automatically 16 byte aligned. In the special case where you wish to align non-vector types to 16 bytes on the stack, you may do so by using a union:

```
//A union that allows memberwise access to a vector float
typedef union
{
    float               f[4];
    vector float        v;
}Float4;
```

Individual stack frames may only be 8 byte aligned, so don't depend on the alignment of the start of the stack frame to correctly align non-vector types to 16 byte bounds. Stack frame conventions are detailed in the **AltiVec Programmers Instructions manual**, Chapter 3.3.

Rename Registers and the Completion Queue

Rename Registers and the Completion Queue often surprise new AltiVec programmers trying their hand at aggressive scheduling of instructions. You may follow all of the advice printed here and still end up with code that doesn't quite work as quickly as you would like. The problem is that the PPC 7400 and 7410 are starved for both vector rename registers and entries in the completion queue. Lack of available rename registers or slots in the completion queue can keep instructions that otherwise are ready to go from entering the execution stage. They will

[2] This author has been known to spend 30 minutes writing vector code and 3 days to write and debug the loop and address arithmetic to drive the unaligned data loads and stores for functions with multiple arbitrarily aligned arrays as inputs and outputs.

typically stall in the dispatch stage for as long as is required until enough resources become available.

Rename Registers

Vector rename registers are temporary buffers used to store results from instructions that have finished execution but have not completed. There are six vector rename registers, six integer rename registers and six FPU rename registers. (The PowerPC 745x processors have 16 of each. For these newer G4's, running out of rename registers is much less of a problem.) For an instruction to be successfully dispatched and to start executing, a rename register must be available for each destination operand specified by the instruction. Once the instruction is done executing, the result is written to the rename register. During the writeback stage of the instruction, the data is copied from the rename register to the destination register. If a subsequent instruction needs the result as a source operand, it is made available simultaneously to the appropriate execution unit. This allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file.

In some cases, on older G4's it is possible that there are more than six instructions in a given unit scheduled to be in flight at a time. In such cases the seventh and later instuctions will stall and wait for one of the other six to complete before it will start. On a 7400 or 7410, you can dispatch one VALU[3] operation and one permute operation to the vector unit per cycle. As VFPU operations can take five cycles to complete, they can consume most of the vector rename registers. If VPERM operations dispatched simultaneously with them, you will run out of rename registers in three cycles. (Due to the nature of the completion queue, instructions only finish in the order that they started, so even though the VPERM operations (permutes) only take a single cycle in principle, in this case they consume a rename register and a completion queue slot each for up to five cycles because the VFPU instruction in front of them in the completion queue hasn't completed yet.) This is typically only an issue with aggressively scheduled code.

Instruction Completion Queue

On the PowerPC 7400 and 7410 up to eight instructions can be "in flight" at any given time. The 7450 can have 16. The currently active instructions are stored in the completion queue. The instructions occupying the completion queue are not limited to just vector operations, they can include other types of operations like integer or floating point instructions, load store operations, etc. The completion queue is a queue in the true sense of the word. The completion unit retires

[3] The VALU is the name given to the combined unit made of the vector floating point unit (VFPU), vector complex integer unit (VCIU) and the vector simple integer unit (VSIU). On a PPC 7400 or 7410, only one instruction can be dispatched to this unit per cycle. On PPC 7450 or 7455, the VFPU, VSIU and VCIU are considered separate for this purpose. You can send instructions to any two of the four vector subunits per cycle on those processors.

an instruction when all instructions ahead of it have been completed, the instruction has finished execution, and no exceptions are pending. This helps guarantee that instructions finish in the order that they were started.

Only two instructions may be retired per cycle on PPC 7400 and 7410. For this reason, there is generally very little acceleration to be gained from simultaneously doing calculations in the VFPU and the FPU at the same time. Between load and store operations, VFPU ops and FPU ops, the completion queue can fill up rapidly because more instructions are dispatched than can be completed per cycle. The 7450 can issue and retire three instructions per cycle.

Branching and Branch Prediction

Branching can be a bit of a problem in AltiVec and elsewhere. An example of a branch might be an if statement:

```
if( test )
    value++;
```

The Sound Sample Conversion 2 code example illustrates problems from branching. Often, when the processor encounters a branch, it may not have enough time to finish evaluating the test before it is time to decide whether to branch or not. As a result, all the processor can do is guess. If it guesses incorrectly, it has to dismantle all operations currently in progress, and restart in the correct place. This is costly.

There are predictable rules about which way the processor will guess. If the branch jumps forward (an if statement) then it is assumed not to be taken. If the branch jumps backward (a loop) then it is assumed to be taken — loops tend to loop more than once. So if you have to add an `if…else` statement to your code, it is best to place the rarely used case after the else, and the most common case after the `if`. These are called static branch prediction rules. On newer G4's these may be ignored in favor of a branch history table, which branches according to how the branch behaved in the past.

If … else ... statements usually concern only a single data stream. (Exception: the `vec_all_*` and `vec_any_*` instructions.) This makes it impossible to write code that can be pipelined or that operates in parallel. As a result, code with a lot of branching in it will operate many times more slowly than branchless code that does the same thing.

The best thing to do about this problem is to find a way to get rid of the branches and write algorithms that work for all possible inputs without special cases. Even if this means that the amount of code triples, it may still be faster. The sample code "Sound Sample Conversion 2" that accompanies this tutorial shows this process. Branching is only a problem in this example in

the scalar code version, but its effect is the same as if you had needed branching in the vector unit. Fortunately, the saturated `vec_packs()` function saves us from having to do any branching in the vector version of the code.

This is the scalar version of the code. It shows the conversion of an array of longs to an array of shorts with clipping. The simple version of the function might look like this:

```
//Clip an array of 32 bit ints down to an array of 16 bit ints.
void Convert( SInt32 *src, Sint16 *dest, UInt32 sampleCount )
{
    SInt32 value;
    while( sampleCount-- )
    {
           value = src[0];
           if( value > SHRT_MAX )
                  value = SHRT_MAX;
           else
                  if( value < SHRT_MIN )
                         value = SHRT_MIN;
           dest[0] = value;
           src++;
           dest++;
    }
}
```

What I have done in the sample code is investigate ways to eliminate the branching. There is a fine tradeoff between branching and branchless algorithms. The branchless variety are often longer, which can make for slower code. On the other hand, branches mispredict, causing the CPU to backtrack.

Real world solutions require testing. In this case, several ways of doing the clip were considered for the integer unit. The simple version looks like this:

```
#define Clip16( value )                  \
    if( value > SHRT_MAX )               \
              value = SHRT_MAX;          \
      else                               \
              if( value < SHRT_MIN )     \
                     value = SHRT_MIN
```

A branchless version looks like this:

```
#define Clip16_2( value )                                        \
    sign = value >> 31;                                          \
    value ^= sign;                                               \
    value = (value | ((32767-value) >> 31)) & 32767;             \
    value ^= sign
```

A version with limited branching and a very short execution path looks like this:

```
#define Clip16( value )                                 \
       if( value != SInt16(value ) )                    \
       {                                                \
              value >>= 31;                             \
              value ^= 0x7FFF;                          \
       }
```

Some of these fail in a very limited set of circumstances around INT_MAX and INT_MIN, but this is not a problem for a mixing buffer. In the end, the version without branching proved to be 4% faster in worst case scenarios in which most of the data needed to be clipped. However in best case scenarios in which less than half needed to be clipped, the short limited branching version was up to 50% faster. Which version to choose is a bit difficult to decide. While it is often said that it is best to optimize for the worst case, 4% is not a very large difference. In addition, the limited branching version has fewer instructions and so should execute much faster when the instructions themselves are not in the cache. Since this particular function is only called once every 11 milliseconds at the most, uncached performance must be considered. When the first pass through the benchmark loop was examined (when the instructions were not in the cache), the shorter version was found to be 20-30% faster.

Load Store Unit (LSU) Peculiarities

Because memory throughput is so important to overall AltiVec performance, it is important to understand how the Load Store Unit functions — what it is good at and perhaps more importantly what it isn't good at. As a basic overview, the LSU has a two or three cycle pipeline into which all loads and stores (including scalar loads and stores), cacheline prefetch instructions and calls to lvsl and lvsr go. This is not to say that loads or stores take only two or three cycles however. This pipeline is really just the front end to massive, partially asynchronous data handling engine that maintains many queues, caches, and tables required to keep track of all the data and mollify the effect of various bottlenecks in the system.

Stores

In some ways, stores are a little bit simpler than loads so we will cover these first. When you issue a store instruction, the store first goes to a three entry finished store queue (FSQ), which then feeds into a 5 entry completed store queue (CSQ). Upon entering the CSQ, the store can be considered complete for all practical purposes that a software developer would care about, though certainly the data still has a long way to go before it reaches RAM. Loads can read the data from the CSQ directly as if they were memory or part of the caches, through a process called data forwarding. When stored data enters the CSQ (sometimes called the store miss queue), its destination address is compared to the address of the block ahead of it. If the two are adjacent

(and in the same cacheline) then they are merged together to make up a single queue entry in the CSQ. This process is called store miss merging and is yet another reason why moving sequentially through memory is a good idea.

When a hunk of data occupying a CSQ slot reaches the front of the queue (CSQ0), (assuming it is cacheable) the cacheline for which it is destined to be written is loaded into the L1 cache from whereever it is, and then the relevant data is replaced by the data in CSQ0. This can take some time, because loading whole cachelines into the L1 can be slow, especially if they are coming from RAM. However, if an entire cacheline is completely overwritten (and is merged to occupy a single entry in the CSQ) then the LSU may issue a kill transaction to kill off the old cacheline load (if in progress) and simply create a new cacheline in the L1 and stick the stored data into it. This can save quite a lot of bus traffic. It will frequently happen automatically if you issue two adjacent vector stores (adjacent both in the instruction stream and the destination address for the data). For this reason, the 7400 user manual recommends that you not waste time with dcbz and just rely on store miss merging to speedily handle adjacent stores. In some cases using dcbz will still be faster. However, it is difficult to recommend use of dcbz, because if the cacheline size changes on future processors, you may end up zeroing more data than you bargained for. Also it is not guarateed that it will continue to be faster on future processors.

Loads

When loads stall, they can stall for a very long time. Such stalls can account for 2/3 or more of the execution time of your function, as we will show later, so it is important to understand how the loads work as well. Even if you can't cut down on the number of stalls, sometimes you can cut down on the length of the stalls, saving you some time.

When a load misses the L1 cache, it enters the processor's Load Miss Queue (LMQ). The older 7400 and 7410 G4's have an eight entry LMQ, and the newer 745x processors have a 5 entry LMQ. Each entry in the queue holds the address of one cacheline to be loaded. The entry will stay in the LMQ until the cacheline is fetched into the caches. During that time, the load instruction will sit in the LSU pipeline in the execution stage and occupy a slot in the Instruction Completion Queue. Since both the pipeline and ICQ are first-in first-out queues, a stalled load can prevent other loads behind it from completing. If they stall long enough the ICQ will become full and no other instruction can enter exection.

There is some amount of finesse required to use the LMQ effectively on the newer PPC 745x class machines. I encourage the reader to read section 6.7.6.5 in Motorola's MPC7450 RISC Microprocessor User's Manual for more information about this. There is some scheduling advantage to loading vectors such that you only load one vector from each cacheline, then go back and load the other vectors from those cachelines.

Load Store interaction

Most of the time loads have priority over stores for attention on the memory bus. This is because stores do not need to occur immediately. Once the data hits the CSQ, it is considered complete, so far as the program is concerned. If necessary, the data there can be forewarded to a load directly from the CSQ. However, if the CSQ becomes full, then there is a priority inversion and the store is pushed out in preference to demand loads. In your app, you will in general find that stores appear to be very inexpensive in **Sim_G4** and in general do not stall. Heavy store activity will sometimes show up as long load stalls in the next loop iteration. Stores can and will stall if you saturate the CSQ however, so there is some reason to limit your loops to processing no more than about 8-10 vectors concurrently, for linear store memory access. After that, performance starts to drop off according to how many additional linear stores are done in a row.

The Big Picture

In order to understand AltiVec you have to put together all of these tidbits of information and synthesize a greater understanding of the vector unit as a whole. This section covers a few special topics in the hopes of being able to help you do just that.

Economies of Scale

The net effect of Alignment and Pipelining is that AltiVec is rarely running at top efficiency unless it can work on 64 bytes or more of data at a time. The reason for this as follows: With unaligned data, handling the edge cases is slow. You may have to load in some surrounding data from the destination buffer, merge it with your results and then save it back again. So these parts of your AltiVec code may be slower than the scalar version of the same thing. You make it back in the middle regions of your data set where alignment costs drop off to nearly zero. In order to be at least 50% not edge case, you need to have four vectors (64 bytes) worth of data. The exception to this rule is when you can pass data into your function by value, rather than memory addresses that have to be loaded.

However, passing vectors in via register or correctly aligning your data does not in itself keep the vector unit happy and well fed. In order to get proper pipelining, it is often necessary to have four (or possibly more in the future!) independent data streams moving through your function simultaneously. The Vector Floating Point Unit (VFPU) has at least a four stage execution pipeline and the Vector Complex Integer Unit (VCIU) has a three stage execution pipeline (four on the 7450). Thus for optimal speed you need to be able to be ready to dispatch four VFPU instructions or three/four VCIU instructions at any given point that do not have any dependencies on one another. Better yet, for the 7450 use both the VFPU and VCIU at the same

time. Having four vector instructions in flight at once means you need to process four (or more) independent vectors at a time through your function. Here again, we need 64 bytes worth of data to get good use of the vector unit. If you write functions that use less data at a time, you should consider declaring them inline in the hopes that they may be able to schedule themselves among the caller's AltiVec instructions.

There are some complex operations like `vec_min(), vec_max(), vec_adds(), vec_rsqrte()` or int FP conversions that are vast improvements over what is to be found in the scalar units. In such cases, it may in fact be cheaper to use the vector unit even if you have only a single element to process.

High Throughput vs. Low Latency Computing

Most programmers think in terms of low latency algorithms — "How can I apply this function to that piece of data in the shortest period of time?" In most cases, the real question you should be asking is "How can I apply this function to my entire data set in the shortest period of time?" The latter approach takes into account the effect of pipelining, parallelization, storage and other factors into the overall design process. This is often described as the difference between low latency and high throughput algorithms.

Because Altivec has comparatively long pipelines, operates on data in parallel, and is often limited by memory bandwidth, high throughput designs are usually much more successful in the vector unit than low latency designs. For example, using the vector unit to multiply a single floating point quantity by another takes four or five cycles. Using the vector unit to multiply four floating point quantities by four others also takes four or five cycles. Using the vector unit to multiply sixteen floating point quantities by sixteen others takes seven or eight cycles. Clearly there is a lot of advantage to handling a lot of data at once!

> Note: It is expected that as processor frequencies get higher, pipelines will get longer. Even though the maximum pipeline length today is five, it is possible that this may grow larger in the future. You can protect your code investment by unrolling loops to handle more data concurrently than is currently necessary. Unrolling to 10 vectors should have little negative effect of the performance of existing processors. After that, the CSQ may start to fill up and you will notice that 32 registers is often not enough.

Suppose you design your functions so that this fact is obvious from the function interface. For example:

```
//rN = vNa * vNb
typedef vector float vfloat;
inline void Multiply(    vfloat v1a, vfloat v2a, vfloat v3a, vfloat v4a,
                         vfloat v1b, vfloat v2b, vfloat v3b, vfloat v4b,
                         vfloat *r1, vfloat *r2, vfloat *r3, vfloat *r4 )
{
    vector float neg_zero = vec_neg_zero();  //function described above
    *r1 = vec_madd( v1a, v1b, neg_zero );
    *r2 = vec_madd( v2a, v2b, neg_zero );
    *r3 = vec_madd( v3a, v3b, neg_zero );
    *r4 = vec_madd( v4a, v4b, neg_zero );
}
```

Anyone calling this function would immediately realize that it is a waste of time to just multiply two floating point vectors together one at a time, and that he would be much better off doing it four at a time. This sort of design motif helps reinforce high throughput programming practices.

> **Note:** It is difficult to return more than one vector from a function in C. Some care must be taken when crafting functions like this so that the return values are passed in register rather than by the stack, which would cause a lot of unnecessary loads and stores to be automatically generated by the compiler. Using pointers or references for return values in an inline function may allow the compiler to optimize away the load store overhead. You may need to disassemble the output to make sure the compiler does the right thing until you are satisfied with its behavior. Another option is to use a C preprocessor macro. However the latter is notorious for causing obscure bugs. It also lacks rigorous typechecking.

There are other sources of overhead that make writing high performance low-latency functions difficult. Vector stack frame overhead is somewhat larger. There is some cost to determining whether it is safe to even use the vector unit. Finally, MacOS X uses a lazy vector register save and restore scheme that in some cases will cause a 1 μs exception to fire when you first use the vector unit in the current thread quantum.[4]

In general, once you pay the cost to get the vector unit fired up you want to make sure you get your money's worth.

When is it Appropriate to use AltiVec?

When I first started using AltiVec, I thought it was only a matter of time before *everything* in my program used the vector unit. This isn't really practical, and maybe not even possible. Most of your program jumps around in memory far too much and has too much branching to run quickly on AltiVec. Where AltiVec really shines is in that 10% of your program that eats up 90% of the CPU. It is useful for routines that work with large amounts of data and for long calculations. These are probably your drawing routines, sound code, large floating point

[4] Some very thorough experiments have shown that lazy register save and restore remains a performance win, despite the added exception cost, because it means the kernel can completely avoid the cost of saving and restoring vector registers on context switches in many cases.

calculations, etc. Coding in AltiVec is a long and painstaking process. You should limit yourself to areas where it is likely to do a lot of good.

A common bit of often cited practical wisdom in programming is that premature optimization is the source of all evil. Because common compilers do not yet generate vectorized code on their own, programming in AltiVec should be considered part of the optimization process. In almost no case is it more readable than normal C code. It certainly takes longer to do right. Thus, you really should not spend time in AltiVec until such time as the bottlenecks in your code are proven and have been shown to be a problem and you know you already have the best algorithm.

This is not to say that you should not plan ahead for AltiVec. Largely those things that you might do, especially aligning your data properly for AltiVec, are quick and easy to do and don't bother scalar code one bit. These typically also will have a beneficial effect on the speed of the scalar code. So, there is no reason not to plan ahead.

Data Organization Suggestions

Altivec has very specific data needs. Because data is aligned in software not hardware, there is a pronounced disadvantage to using unaligned data. If possible, attempt to ensure that every vector is 16 byte aligned. Even if you are not sure you are going to use AltiVec, it is often a good idea to align your data anyway. It is harder to retrofit the changes into your application later, and good alignment almost never hurts scalar code.

AltiVec likes to have its data all in one place. This is because more often than not the speed of AltiVec functions is limited by memory throughput. If your data is scattered throughout memory, you will be loading one cacheline per vector on average. If your data is all in once place you will be loading one cacheline per every two vectors on average. More importantly, the translation lookaside buffer (TLB, part of the unit that maps memory addresses to hardware locations) is up to 256 times less likely to cause a 150+ cycle stall with linear memory reads than for random memory reads. Here again, the scalar unit can benefit from these modifications as well, so even if you are not sure you will use AltiVec for a function, it doesn't hurt to plan ahead. In some cases, large arrays present a problem for object oriented code. In these cases you have to evaluate your opportunities for parallelism within OO code. Large segments of OO code only need to operate on a single data thread, so wouldn't benefit from SIMD much anyway. Only when you can operate on multiple data in parallel is Altivec worth the effort.

Finally, the ordering of elements within a vector is often very important. If the elements in your vector represent different types of quantities, then typically you will find your function growing very complicated with a lot of permute operations, redundant calculations and lost

opportunity for parallelism. You may only get a factor of two or three speed gain with your data organized this way.

> Example: a vector full of 32 bit pixels has four different kinds of elements in it for the four different color channels. Often this leads to overly complicated code to handle the alpha channel differently from the red, green and blue channels.

If the vector holds 4, 8 or 16 of the same thing, then the math is much more straightforward. You are likely to get a full factor of N rate acceleration over scalar code or more, where N is the number of elements in the vector. Functions that use uniform vectors are typically easier to read and write because they look just like the scalar code. They usually take better advantage of pipelining within the processor. They rarely require the use of the permute unit at all. The constants that they use tend to be simpler and more easily generated. You almost never do redundant work.

> Example: reorganize your data into a vector of 16 alpha channels, a vector of 16 red channels, a vector of 16 green channels, and a vector of 16 blue channels. As a token of good will to the video hardware we note that just before you send your data to a display device, you can use the `vec_mergeh` and `vec_mergel` instructions to interleave the data to the more conventional data layout fairly quickly.

This topic is covered extensively on the Apple AltiVec web site:

http://developer.apple.com/hardware/ve/simd.html
http://developer.apple.com/hardware/ve/data_handling.html

Memory Speed is Often the Problem

The biggest speed impediment to PowerPC performance today is memory overhead. This is increasingly true as processors move to higher and higher bus speed ratios between processor clock speed and motherboard clock speed. Whereas a 68k Macintosh might have had its processor running at the same clock speed as its motherboard, modern PowerPC machines might be running four, five, six or even seven times as fast as their memory subsystems. What this means is when there is a cache miss, you may have to wait for tens or hundreds of CPU cycles for the memory systems to catch up to you.

The thing to know about the G4 is that the L2 (and L3) caches serve as a victim cache – data only comes to be in the L2 or L3 caches after being cast out of the L1 (or L2) cache. Data has to be moved to the L1 cache before it can be loaded into register. This means that every piece of data that you use has to be loaded in the slow way at some point, and if you only touch a piece of data once or once in a while, it will almost always be loaded in the slow way. Loading a 32 byte cacheline from L2 takes from 10-15 cycles. Loading a cacheline from RAM to L1 takes about 35-40 cycles on my G4/400, provided that the page is in the TLB. One cacheline is two vectors worth of data. If all you do is add those two vectors together (as little as 1 cycle), then during the other 39

cycles your code will do nothing. If the prospect of having your code running at $1/40^{th}$ of its expected speed bothers you, then this section is for you.

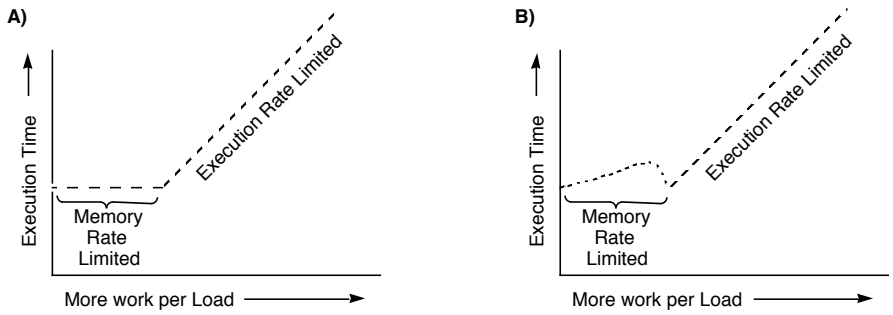
Because the speed bottlenecks in the processor have changed so much over the last few years, it stands to reason that some aspects of code optimization have to change too. I am trying to introduce this idea gradually because I know that a lot of very experienced programmers are going to be very resistant to the idea that some of their favorite code optimization techniques rely too heavily on memory, perhaps slowing down the code rather than speeding it up. If you are one of these programmers, I urge you to keep reading. Some optimization paradigms presented here may be new to you. I think you will find them useful:

The Fastest Functions are Those That Do More

If the speed of your function is limited by the speed of memory, you have approximately 40 cycles of time to work on each 32 byte chunk of uncached data. (This is the time it takes to load in each cacheline.) If you don't use that time, then it is wasted. You will likely spend the rest of those 35-40 cycles stalled waiting for more data to appear. This is easily verified by profiling a memory intensive function using **Sim_G4**. Chances are you will see some very long stalls in `lvx`. Compare the amount of time spent in the stall to the amount of time processing data, and you may find that most of your time is wasted.

It is a common programmer instinct to break down complex problems into simple ones. Resist that urge in AltiVec code that touches lots of memory. 35-40 cycles is a very long time! You will have to work quite hard to use all of it processing only 32 bytes of data. Remember that these extra cycles have already been spent for you, so if you don't use them, they are gone. If you can replace any work anywhere else with code here, you will gain that much more speed because work done here is "free", you save a load/store pair and perhaps an additional cache miss.

Surprisingly, doing more work on a set of data, even if it is totally gratuitous work, often will take no addition time and can in some unusual cases even speed up the function. *What?!* This is a rather peculiar side effect of some features of the older G4's. It is common wisdom that the performance of a function looks something like shown in example **A** below.

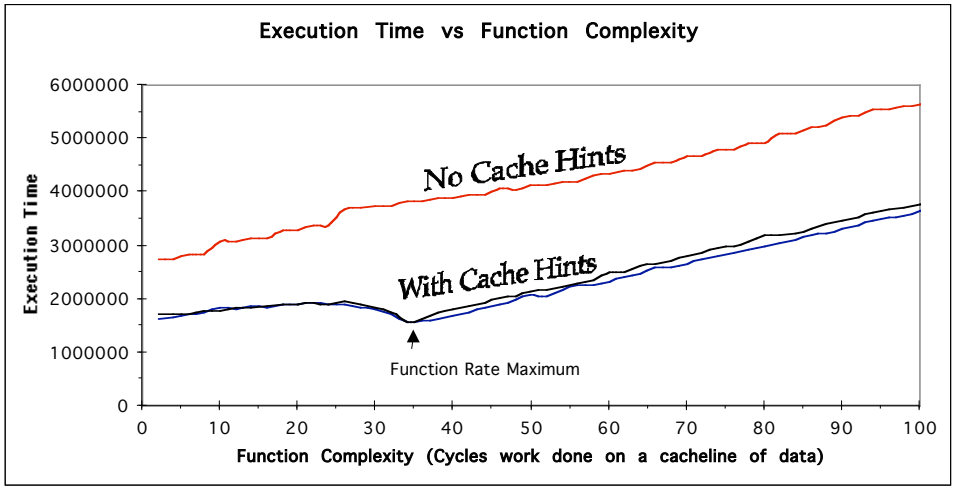


A) The execution time of a function is dependent on how much work it has to do, unless some other factor (e.g. memory speed) becomes the bottleneck. **B)** Instruction completion queue stalls make the problem worse.

Actually, the line shape is more like **B** on an older G4. I suspect this is due to some interaction down inside the LSU, but I must admit that I have not investigated it further.

> On a PPC 7450 or 7455 processor, the lineshape looks a bit more like the A case above. Apple Computer's AltiVec website shows a similar study on a PowerPC 7450 processor: http://developer.apple.com/hardware/ve/performance_memory.html

Why is this interesting? When the data processing complexity gets sufficiently large that memory speed is no longer the dominant rate limiting factor, memory stalls disappear. Provided that cache hints are used, this means that the memory unit and the rest of the processor can operate in parallel rather than in series. The function runs faster in this case, because the time required to load the data is removed from the execution time of the function. The function can be thought of as running at maximum speed at the point where the time it takes to load a cacheline from RAM is almost exactly the same as the time it takes to process it.



Note that the layout of this graph has a tendency to flatten out the effect a bit and may make it appear less significant than it really is. The actual speed difference between the case at 35 add ops (1.5 MTicks) and the slower case at 25 add-ops (1.8 MTicks) that does less work represents about a 20% speed difference.

Above are the execution times for a benchmark program that adjusts independently the amount of data processing work done per cacheline loaded, and also the size of the vector prefetch, in case that had any effect. The data shown is for evaluation times over a very large data set, that does not fit in the caches at all. This data was collected on a G4/400 (PPC7400).

The red line represents what happens if you do not use cache instructions. Because there is no prefetch, loads and data processing have to happen serially. Because we can't do much in parallel, memory overehead and data processing times are additive and there is no flattening off effect.

The blue and black lines represent the best and worst case when using `vec_dst()` to prefetch data in blocks sized between 32 and 512 bytes. (I ran 16 such cases. They all fall between these two lines.) In all such cases, the function is moving at its fastest when 35 cycles of processing time is devoted to the data, the same amount of time it takes to load in the data. It is particularly reassuring that the part of the curve that corresponds to CPU rate limited code can be extrapolated to run through nearly through zero. That shows that memory overhead for complex functions using `vec_dst()` is near zero. We also note that the version that does not use `vec_dst()(red)` and the versions that do (black and blue) have very nearly the same slope. Thus, it appears, the complexity of the function is the prime rate determining factor with highly complex functions. This is exactly what you want!

> Note: Unfortunately, the exact position of the "sweet spot" where the function is cycling at its fastest on an older G4 likely varies from machine to machine. This is because memory load times are dependent on how the memory systems are set up, particularly the bus speed ratio between motherboard and CPU, the nature of the RAM used, and other factors. Therefore, it is probably not a good idea to just add a bunch of noops to your function to speed it up. You can still take advantage of this effect however, provided you can find some real work to do to fill the extra time. As long as you are to the right of the sweet spot on the graph, your function is running at near 100% efficiency with near zero memory overhead. You really can't ask for anything more!

How does this finding shed light on your code? Unless they are very complicated functions, chances are most or all of your AltiVec functions do much less than 35 cycles worth of work per cacheline data. 35 cycles is a very long time. 35 cycles is enough time to calculate the dot product of two vectors with 112 elements in each! Since we can process nearly a kilobyte of memory in the time it takes to load 32 bytes, it should be apparent that it really requires a *very* complex function to be slower than RAM. Thus, the more work you can do in your function, the better off you will be! Significantly, it should be observed that code written in this fashion will work equally well no matter where the data is (except for page misses), making it a great example of a place to use the transient cache instructions and LRU loads and stores.

Well yeah, but what about the caches? What about them? Sure the caches are fast, but the data will only be in the cache the second time you use it within a short time. You *will* be paying the price

of a slow trip to RAM at some point no matter what you do. Why not do all the work on the data the first time you load it into register, and then flush it back out of the cache immediately using the use of the LRU instructions so that it doesn't displace frequently needed stuff? Save the caches for things that really need it.

The Fastest Algorithms Are Often the Ones That Use Less Data

If based on the above evidence you accept that memory overhead is the rate limiting factor for your AltiVec code most of the time, then it almost goes without saying that the fastest algorithms are the ones that use less data. However, did you consider the implications of this statement? What your mother told you about writing fast code is quite possibly no longer true! For example…

Lookup Tables Are Not Fast

Lookup tables, especially large ones, are not as fast as many would like to believe for a number of reasons. Most obviously, if you incur one cache miss accessing your lookup table, you can lose 40-250 cycles waiting for the data to load. That is a HUGE amount of time! Think of what you could have done with it.

Consider also that if you are using a lookup table, you are hopefully using it to lookup a lot of data. (A rarely used lookup table is almost guaranteed to not be in the cache, meaning abysmal performance because of lots of cache misses.) Functions that use a lot of data have high memory throughput needs. This means that you are probably already memory rate limited just loading in all the data that you want to use to index the looup table. Recall that the execution time data **shown above** showed that memory bound code has about 35 cycles of dead time to fill with calculations. You could use that time to do the brute force calculation instead of the lookup and avoid further taxing the memory systems reading data from your table.

It is also hard to look up data in parallel in the vector unit. Often you have to do it one item at a time. Why not do a brute force calculation for 4, 8 or 16 items at a time?

Finally, every time you load data in for a lookup table, you displace something else from the cache. Whatever that is, chances are it will have to be loaded back in later. Doing a brute force calculation will preserve that data in place meaning that code elsewhere in your application will run more quickly. Brute force calculations can be fast, free, and more accurate. Your data caches will thank you.

The only lookup tables likely to do you any good relative to brute force methods are the ones that save a LOT of calculation (e.g. CRC-32), and those lookup tables that are so small you can preload them into register and then process a lot of data. You can do a nice small fast lookup

table with `vec_perm()`, for example, but this approach seems to limit you to tables of perhaps 256 entries and extra work is required if the table cells are not 8 bits in size.

If you are still doubtful lookup tables are slow, I suggest you run some experiments. It would be helpful to do it on a bottleneck function in place in the app so that the full effect of displacing other needed data from the caches impacts the performance of your app and is measurable.

Longer Functions Are Slower

Data is not the only thing that needs to be loaded into the cache before it can be used. Instructions need to be loaded too. The instructions are loaded eight at a time, and the speed penalty for each such load is once again usually around 35 cycles. Thus, each uncached instruction that your function has to load has an average memory overhead of around four to five cycles. Since most instructions only take one cycle to execute, more often than not, the fastest uncached code is the shortest code. **Thus, for rarely used code paths, there is a very good reason NOT to attempt to optimize them, since optimized code is often longer**. When executing rarely used code, there is a lot of extra time spent standing around, which could be used for other things. If you habitually code funny math "shortcuts" using *many* short instructions to avoid single multi-cycle instructions like integer multiplication or division, you may be better off not doing so. Likewise, using large switch statements just to avoid a few cycles worth of work are likely to be counterproductive. Unoptimized, your code will be easier to read and shorter. In addition, keeping your code small means that it displaces less other code from the caches.

One clear exception to the rule is any code in a loop. Loops get very good code reuse and have great temporal locality. The first time you read through a loop, it will execute as uncached code (if it is uncached) but after that it will be running at full speed. So if you are going to make gratuitous optimizations to rarely executed code, save it for loops.

Be Careful of Constants and Globals

Most programmers new to AltiVec make copious use of variables that have to be loaded in from memory each time the function is called. These may be globals or static constants defined like this:

```
vector signed char myConst = (vector signed char) (23);
```

Perhaps you have a global used in a tight loop. Normally one might think that the compiler will do the smart thing and load the global into register and then use the copy in the loop, but it can't. The reason is that some other thread or interrupt level task might change the global, and so it has to be loaded in every time. **Always explicitly load in globals, constants and other items that have to be loaded from memory into a variable local to the function, and use the local variable**

in your function. This will enable the compiler to avoid any excess memory overhead associated with redundantly loading in data over and over again.

> This is even sometimes found to be true of the high order index for 2D arrays. If the compiler cannot determine that the array pointers are unchanged by your function, the pointer to your current row may be redundantly reloaded each time through the loop.

Almost Everything in AltiVec Is A Blitter

When it comes right down to it, most functions that can be accelerated for AltiVec move large quantities of data from A to B, possibly changing it along the way. Since the time it takes to load and store the data is usually the rate-limiting factor for these operations, such functions are to most standards simply blitters — functions for rapidly copying data from place to place. I'm not sure if this truism really means anything. However, it might get you thinking in the right direction for how to speed up your code.

Useful Tools

Here are a couple of useful tools for sorting out AltiVec performance issues:

Debuggers

There is a version of **MacsBug** on **Apple's AltiVec site** that has **AltiVec dcmds**. If you use MacsBug, you will find this handy.

GDB is AltiVec aware. To display the contents of vector register 22, use:

```
print $v22
```

Sim_G4 and Amber

Sim_G4 and **amber** enable you to examine G4 code performance on an instruction by instruction basis on MacOS 9. This is invaluable for locating real world causes for stalls. **Sim_G4** is available for MacOS X as part of the **CHUD SDK**. After installation, it is usually to be found in `/usr/local/bin.` (However, it may move at some point in the future.) The meaning of the various symbols used in Sim_G4 is discussed here:

http://developer.apple.com/hardware/ve/performance.html

An example where SimG4 is used to analyze the behavior of a piece of code is presented later in the section entitled, "**Special Topic: Optimization**".

> Note: If you used Sim_G4 on MacOS 9, that version and pitsTT6Lib are now obsolete. The CLI based software that ships with **CHUD** should be used instead.

Sim_G4 requires a TT6 trace in order to work. Amber is utility that you use to generate that. (It also lives in `/usr/local/bin.`) It is impractical to trace more than about 100,000 instructions, so in typical use, you will need to signal amber to start sampling just before your function is called and stop soon after it finishes. One way to do that is to insert illegal supervisor level instructions immediately before and after your function:

```
#if defined( __GNUC__)
	#include <ppc_intrinsics.h>
#endif

int RunTest( void )
{
	int i;
	i = __mfspr( 1023 );	//illegal instruction

	CallYourFunctionHere();

	i |= __mfspr( 1023 );	//illegal instruction

	return i;
}
```

Amber will intercept the illegal instructions and use them as signals for when to start and stop tracing. To actually collect the trace, run amber from the terminal:

```
usr/local/bin/amber -f 6 -i /path/to/your/application
```

If your application is a CFM application, you can use this instead (all one line):

```
/usr/local/bin/amber -f 6 -i
/System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp
/path/to/your/application
```

This will produce one or more trace_### directories. Inside each should be a TT6 file to use with Sim_G4. If you don't want to add illegal instructions to your app (or can't), you can also use the –b and –e flags to set the beginning and ending instruction addresses at which to sample, instead of the –i flag. You can also use the –x flag to set a maximum instruction count. This is useful for keeping traces down to a reasonable size.

SimG4 takes a variety of flags. I find the following two flavors the most useful:

```
/usr/local/bin/simg4 -sp simg4.txt -st 0 -sw 80 -r warmup=1 < trace.tt6
/usr/local/bin/simg4 -sp simg4V.txt -st 2 -sw 80 -r warmup=1 < trace.tt6
```

This will produce a `simg4.txt` output file. The first uses a horizontal scroll pipe. The second uses a vertical scroll pipe. The `–r warmup=1` flag prewarms the caches in the simulator so that it doesn't include lots of cacheline misses in the simulation. If you like train wrecks, you may admire the trace with this option missing. The –sw flag sets the width of the text display.

At the end of this process, you will be presented with a cycle by cycle accounting of what happens in a PowerPC 7400 with your code. The text output would look something like this:

```
85 D*F#       R E  |   72:(  77)* fadds   F11,F10,F29       |   73:(  77)# bc+    16,0,0xffdc       | Maximum Allowed Dispatched
86 D F D*       E  |   74:(  78)* lfs     F12,0x700(R9)     |                                       | Double Load/Store
87 D F E D*     F  |   75:(  78)* lfs     F31,0x704(R9)     |                                       | FPU Unit Busy
88 E F F E D*D# R  |   76:(  81)* fadds   F13,F11,F12       |   77:(  83)# lfs    F30,0x708(R9)     | Maximum Allowed Dispatched
89 E F F F D E D*D# |   78:(  86)* lfs     F29,0x70c(R9)     |   79:(  86)# addi   R9,R9,0x10        | Maximum Allowed Dispatched
90 F F F F D F E F  |                                        |                                       | CB Full
91 R R F F E F F F  |                                        |                                       | CB Full
92 D*  R R E F F F  |   80:(  87)* fadds   F0,F13,F31        |                                       | Maximum Rename FPR's Reache
93 D       F F F F  |                                        |                                       | FPU Unit Busy
94 E D*    R R F F  |   81:(  88)* fadds   F10,F0,F30        |                                       | FPU Unit Busy
95 E D         R R  |                                        |                                       | FPU Unit Busy
96 F D              |                                        |                                       | FPU Unit Busy
97 R E D*F#         |   82:(  89)* fadds   F11,F10,F29       |   83:(  89)# bc+    16,0,0xffdc       | Maximum Allowed Dispatched
98   E D F D*       |   84:(  90)* lfs     F12,0x700(R9)     |                                       | Double Load/Store
99   F D F E D*     |   85:(  90)* lfs     F31,0x704(R9)     |                                       | FPU Unit Busy
```

Each row is a CPU cycle. The first column (populated by lots of single character symbols, such as D, E, F and R) shows the progression of each instruction through the processor's pipelines. (D means dispatch; E means execution; F means finished execution and waiting to retire; R means retiring.) The second column shows the first instruction that entered dispatch that cycle. The second column shows the second instruction that entered Dispatch that cycle. The last column is a list of excuses why the processor did no do more that cycle.

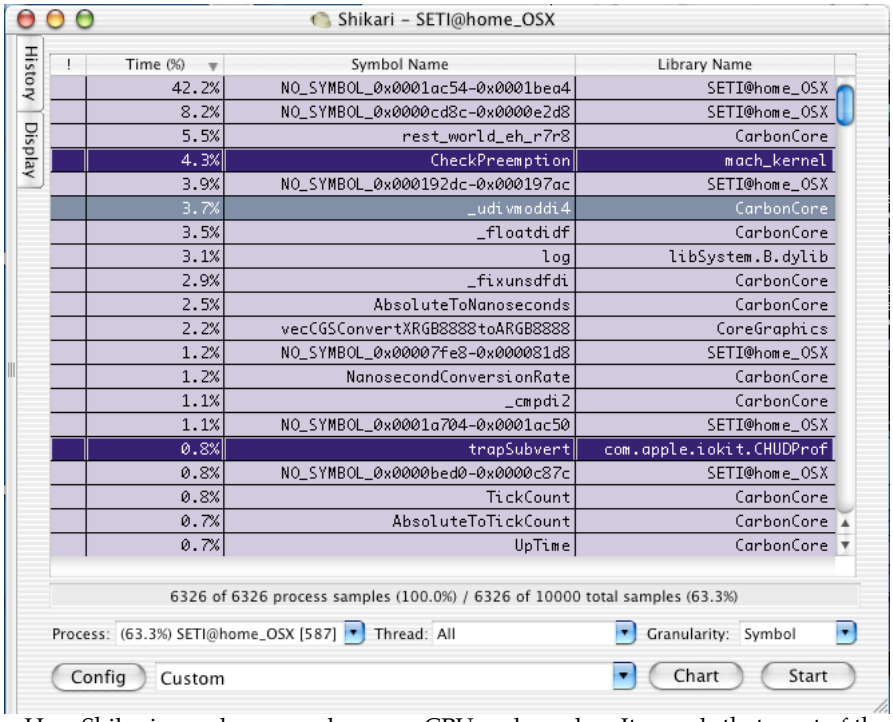
Shikari and MONster

The G4 processor has a suite of performance monitor counter registers, that can count the number of times selected events happen. For more information on these, please read the Performance Monitor Chapter in the **User Manual** for your processor. Sample events might be CPU cycles, L1 cache misses, AltiVec loads, FPU dispatch stalls, etc. Since these counters operate in hardware, they make for a powerful way to characterize the performance of your application without adversely affecting its operation.

Apple has released a suite of tools that let you take advantage of these registers in the form of the Computer Hardware Understanding Development SDK. This contains close to a dozen applications, including **Sim_G4** and **amber**, described above. MONster is handy for using multiple performance monitor counters in parallel. This would let you do things like count both AltiVec stores and the number of merged store misses to get an estimate of how well you are able to take advantage of the **store miss merge mechanism**.

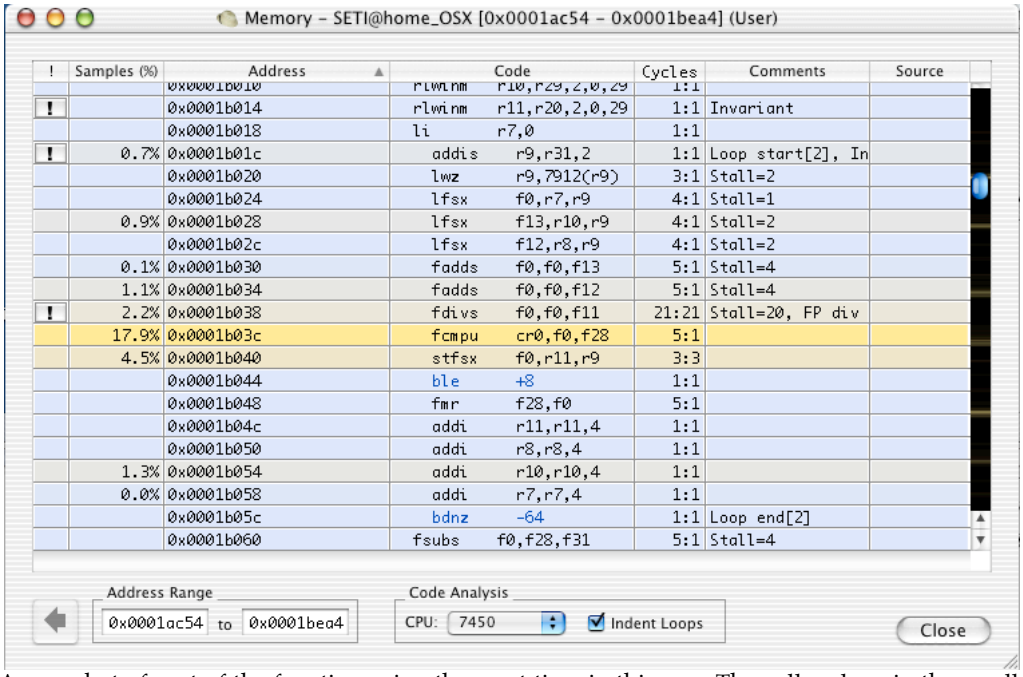
In addition, there is Shikari. Shikari is a sampling application that will stop your program every so many times an event happens and record which instruction was executing. This lets you

characterize where these events are most likely to happen in your app. If you are interested in L3 cache misses, you can set Shikari to sample on L3 cache misses, run your app, and get a statistical breakdown of which functions in your app are taking L3 cache misses. Similarly you can have the sampler just sample on CPU clock ticks. This will give you a time based sampling that will provide information similar to what you would get out of a traditional profiler, that tells you which functions are using the most CPU time. If you double click on the function, you can get a breakdown on an instruction by instruction basis, which should help you decide which parts of your functions are taking most of the CPU time and potentially why. Sampling based on various performance monitor counter events will help answer any remaining questions about why certain things are slow.



Here Shikari samples a popular spare CPU cycle-sucker. It reveals that most of the CPU time is spent in just one or two routines and also that this app is spending at least 16.4% of the CPU polling on `TickCount()` and `UpTime()` and exercising associated time conversion functions and their progeny.

You can double click on one of the functions to get a detailed view of the assembly and which parts of the function are using time. If we look at the function responsible for much of the time above, we see that the CPU time is localized in just a few parts of the code, and that there is quite a bit of time lost to division and dynamic data dependency stalls:


Memory - SETI@home_OSX [0x0001ac54 - 0x0001bea4] (User)
! Samples (%) Address Code Cycles Comments Source
0x0001b014 rlwinm r11,r20,2,0,29 1:1 Invariant
0x0001b018 li r7,0 1:1
0.7% 0x0001b01c addis r9,r31,2 1:1 Loop start[2], In
0x0001b020 lwz r9,7912(r9) 3:1 Stall=2
0x0001b024 lfsx f0,r7,r9 4:1 Stall=1
0.9% 0x0001b028 lfsx f13,r10,r9 4:1 Stall=2
0x0001b02c lfsx f12,r8,r9 4:1 Stall=2
0.1% 0x0001b030 fadds f0,f0,f13 5:1 Stall=4
1.1% 0x0001b034 fadds f0,f0,f12 5:1 Stall=4
2.2% 0x0001b038 fdivs f0,f0,f11 21:21 Stall=20, FP div
17.9% 0x0001b03c fcmpu cr0,f0,f28 5:1
4.5% 0x0001b040 stfsx f0,r11,r9 3:3
0x0001b044 ble +8 1:1
0x0001b048 fmr f28,f0 5:1
0x0001b04c addi r11,r11,4 1:1
0x0001b050 addi r8,r8,4 1:1
1.3% 0x0001b054 addi r10,r10,4 1:1
0.0% 0x0001b058 addi r7,r7,4 1:1
0x0001b05c bdnz -64 1:1 Loop end[2]
0x0001b060 fsubs f0,f28,f31 5:1 Stall=4
Address Range
0x0001ac54 to 0x0001bea4
Code Analysis
CPU: 7450
Indent Loops
Close

A snapshot of part of the function using the most time in this app. The yellow bars in the scroll bar reveal highly localized regions where the vast majority of time is spent. We are looking at one such inner loop, in which there is an initial stall between 0x1b020 and later loads, probably to index into a 2D array. There are also a number of dynamic data dependency stalls in the ensuing fadds operations, because they are serially dependent on each other. (Each add requires the result from the add before it, so we have to wait for each add to finish before we can start the next. Such code loses all advantages from pipelining, and on a PPC 7450 is only operating at 20% of the speed it could be.) Finally, there is an expensive division operation followed by a floating point compare operation (with accompanying stall and potential for branch miss). Much of the time lost in this loop could be reclaimed if it was unrolled a little. It is a good candidate for vectorization. We can see that the loop counters are advancing by one float each iteration.

Compilers

There are more and more AltiVec enabled compilers available. I don't have experience with all of them, so I will just discuss briefly **CodeWarrior** and **Project Builder/gcc** here. Whatever compiler you use, it is recommended that you look at the disassembled output that it produces to verify that you are getting the sort of rigorously optimized code that you intended.

CodeWarrior

To set up a CodeWarrior project for go to the project preferences panel, scroll down to the "Code Generation" section and click "PPC Processor". Make sure the "AltiVec Programming Model" box is checked. Also make double sure that the "Generate VRSAVE Instructions" box is also checked. If you don't generate VRSAVE instructions on MacOS, VM paging and interrupts and preemptive threads will squash the contents of your vector registers at random times. This will lead to random errors in your program.

Using assembly within C code in Codewarrior is easy. (You frequently need to do this to use cache instructions like dcbz.) Many asm functions have inline assembly functions named with a double underscore followed by the name of the instruction:

```
// Zero the 32 byte aligned cacheline that contains ptr
__dcbz( ptr, 0 );

// Rotate var left 8 bits and insert bits 0-7 into result
result = __rlwimi( result, var, 8, 0, 7 );
```

You can also code assembly in code blocks within C functions. This occasionally has to be done when an instruction doesn't have a inline assembly macro:

```
//Convert a float to a unsigned char with saturated clipping
inline UInt8 ClipNConvertDouble( register double value )
{
    const double conversionFactor = (double) ULONG_MAX / (double) UCHAR_MAX;
    const double signedToUnsignedOffset = (double) LONG_MIN;

    union
    {
            double  d;
            struct
            {
                    UInt32  junk;
                    UInt8   loByte;
            }i;
    }temp;

    //Rescale the value to be between LONG_MAX and LONG_MIN from 0...255
    value = value * conversionFactor + signedToUnsignedOffset;

    //Conver to integer, with built in clip
    asm{ fctiw      value, value }

    //write the result to the stack, so we can load it into a GPR
    temp.d = value;

    //Load in the -128...127 result and correct back to 0...255
    return temp.i.loByte + 128;
}
```

…and of course, you can write functions entirely in assembly:

```
//Read the current time from the processor's Time Base Registers
//(arbitrary units)
asm UInt64 ReadTBR( void )
{
    loop:
           mftbu  r3            //load from TBU
           mftb   r4            //load from TBL
           mftbu  r5            //load from TBU
           cmpw   r3,r5         //see if 'old' == 'new'
           bne    loop          //loop if carry occurred

           blr                  //return
}
```

ProjectBuilder and/or gcc

Project Builder is Apple's IDE, part of their suite of tools for writing CLI, Carbon and Cocoa applications. It makes use of Apple's version of GCC, the Gnu C Complier to do the actual compilation of code. Configuring Project Builder to use AltiVec is fairly straightforward. Select "Edit Active Target" in the Project Menu. Click the GCC Compiler settings line at left. Add the following to the "Other C Compiler Flags" text box: -faltivec. For GCC on the command line, just add the –faltivec flag. There are no libraries or headers to include.

http://developer.apple.com/hardware/ve/tutorial.html

Generally, performance improvements between no optimizations and optimization level 3 with GCC are quite pronounced, and may be as much as a factor of four.

> ▲ CAUTION: In Project Builder, the optimizer may be turned off for the development target. ▲

Using assembly in PB/gcc is a little bit trickier than with CodeWarrior. The basic asm device can be cryptic. The details of which are covered here:

http://www.devworld.apple.com/techpubs/macosx/DeveloperTools/Compiler/Compiler.3c.html
http://gcc.gnu.org/onlinedocs/gcc_5.html#SEC104

This example macro allows you to use `__dcbz( ptr, offset)` as in the CodeWarrior example above:

```
// Zero the 32 byte aligned cacheline that contains ptr
inline void __dcbz( void *buffer, int buff_offset )
{
    __asm__ __volatile__ ("dcbz %0,%1"
        :
        : "b%" (buff_offset), "r" (buffer) : "memory" );
}

//Read the current time from the processor's Time Base Registers
//(arbitrary units)
unsigned long long ReadTBR( void )
{
  __asm__ volatile
    ("0: mftbu r3\n"
     "mftb r4\n"
     "mftbu r5\n"
     "cmpw r3, r5\n"
     "bne- 0b");
}
```

Starting with MacOS X.2 (Jaguar) Apple released a header called "ppc_intrinsics.h" that has Codewarrior style inline asms predefined using this methodology for you to use. Simply:

```
#include <ppc_intrinsics.h>
```

in your code before using these things. This header should also work with other flavors of GCC.

Apple compiles its precompiled headers with `–faltivec` off. This causes the precompiled headers to fail if you turn the flag on, leading to some slow compile times and lots of warning messages. If you spend most of your time working with apps that use `–faltivec` and you have this problem, then you will likely benefit from recompiling your precompiled headers with `–faltivec` on. This can be done from the terminal as follows:

```
sudo fixPrecomps –force –precompFlags –arch ppc –arch i386 -faltivec
```

To undo the changes, do:

```
sudo fixPrecomps –force –precompFlags –arch ppc –arch i386
```

> Check the end of the post install script on Apple's developer tools installer for what Apple does to precompile these when it installs the compiler on your machine.

GCC 3.x based compilers usually generate much faster AltiVec code than older 2.9.x based compilers.

Absoft FORTRAN

Absoft has AltiVec support in their FORTRAN compiler. I've never used it, but they have some information for would-be FORTRAN vectorizers on their website, including how to do things like link to the vecLib framework.

Special Topic: Writing Code that Runs on Both G4 and Earlier Processors

It is possible to write a single application that uses AltiVec and still works on both G4 and earlier processors. There are some special issues that must be dealt with correctly however. Otherwise, there is a good chance that your pre-G4 clients are going to stumble on some AltiVec code and crash.

Determining the Runtime Architecture

The first thing you need to do is determine whether you are running on a G4 or not. Apple provides some Gestalt functions to help you. It is better to use the one that tells you whether or not AltiVec is supported, rather than relying on the processor ID or some other method. Here is some sample code:

```
//Return true if it is safe to use the AltiVec unit
Boolean IsAltiVecAvailable( void )
{
    long  processorAttributes;
    Boolean result = false;

    OSErr err = Gestalt(gestaltPowerPCProcessorFeatures,
                        &processorAttributes);
    if( err == noErr )
          result = (1 << gestaltPowerPCHasVectorInstructions) &
                processorAttributes;

      return result;
}
```

For Mach-O based executables, you can do similar things through sysctl():

```
#include <sys/sysctl.h>

Boolean IsAltiVecAvailable( void )
{
    int selectors[2] = { CTL_HW, HW_VECTORUNIT };
    int hasVectorUnit = 0;
    size_t length = sizeof(hasVectorUnit);
    int error = sysctl(selectors, 2, &hasVectorUnit, &length, NULL, 0);

    if( 0 == error )
          return hasVectorUnit != 0;

    return FALSE;
}
```

If `IsAltiVecAvailable()` returns `true`, then you can use AltiVec in your program, otherwise, you must avoid it.

Using Plug-ins

The easiest, safest way to write code that works on both G3 and G4 is to move the G4 specific code off into a plug-in. You can have a scalar plug-in that contains the scalar version of a set of functions, and an AltiVec plug-in that contains the AltiVec version. Simply load in the appropriate plug-in in your application initialization phase. Writing a plug-in is fairly straightforward. Set the compiler to create a shared library instead of an application in the project preferences panel. Link various external libraries (e.g. CarbonLib) to it as you would a normal application as needed. In our example, we will write a plug-in that provides these two functions:

```
void BlitTransparent16( char *source, char *target, Uint32 pixelCount );
void BlitTransparent32( char *source, char *target, Uint32 pixelCount );
```

You need to make a .exp file to tell the compiler which functions to make publicly available. (Make sure to set the Linker to use a .exp file to decide which symbols to export.) ".exp" files are easy:

```
#
#  BlitterLib.exp
#
#  Exports file for the shared library
#  Just list the names of the functions you want to make available
#


BlitTransparent16
BlitTransparent32
```

Finally, you just write the two functions for the scalar units and compile them into a plug in, then write the AltiVec versions and compile them into a different plug-in. Do not link your application against the plug-in. We are going to load it manually.

In your application, first test to see if AltiVec is present. If it is, load in the vector plug-in, otherwise load in the regular one. Loading in plug-ins using the **Code Fragment Manager** is fairly straightforward:

```
//Our global function pointers declared with the same type as in the lib
//After the plugin is loaded, these will contain function pointers to the
//correct version of the function for the current runtime environment.
OSErr (*BlitTransparent16)(void *src, void *dest, Uint32 count) = NULL;
OSErr (*BlitTransparent32)(void *src, void *dest, Uint32 count) = NULL;
CFragConnectionID gBlitterLibID = 0;

OSErr LoadBlitterPlugin( void )
{
    Ptr                 mainAddr = NULL;
    Str255              errName;
    OSErr               err = noErr;
    CfragSymbolClass    symClass;

    //Load the correct library according to whether AltiVec is
    //available or not. Simply pass the name of the library.
    if( IsAltiVecAvailable() )
          err = GetSharedLibrary(    "\pBlitterLib AltiVec",
                                     kPowerPCCFragArch,
                                     kReferenceCFrag,
                                     &gBlitterLibID,
                                     &mainAddr,
                                     errName);
    else
          err = GetSharedLibrary(    "\pBlitterLib Scalar",
                                     kPowerPCCFragArch,
                                     kReferenceCFrag,
                                     &gBlitterLibID,
                                     &mainAddr,
                                     errName);

    if( noErr != err )
          return err;
```

```
   //Load in our function pointers one at a time by name
    err = FindSymbol(   gBlitterLibID,
                        "\pBlitTransparent16",
                        &((Ptr) BlitTransparent16),
                        &symClass);
    if( noErr != err )
           return err;

    return FindSymbol(  gBlitterLibID,
                        "\pBlitTransparent32",
                        &((Ptr) BlitTransparent32),
                        &symClass);
}

//Call only when finished using the plug-in functions
void UnloadBlitterPlugin( void )
{
    if( 0 != gBlitterLibID )
            CloseConnection(&gBlitterLibID);
    gBlitterLibID = 0;
    BlitTransparent16 = NULL;
    BlitTransparent32 = NULL;
}
```

The plug-ins need to be in the CFM search path. In Carbon, you can also use the routines in CFBundle.h for loading code fragments. These have the advantage that they also work for OS X mach-o frameworks as well. However, calling functions in libraries adds a bit of extra overhead due to glue code. Calling one of these functions repeatedly in a tight loop might be costly.

As a Single Monolithic Application

The down side of plug-ins is that there are about 5 instructions worth of extra overhead for each function call. It's not much, but if your function only has 5 instructions in it, then this could add up. It is possible to write a single application that does everything without plugins, provided that you follow a few rules.

Mixing AltiVec and Scalar Code in the Same Function:

First of all, you can **not** mix AltiVec and scalar code in the same function, **even if** you do a check before entering the AltiVec code:

```
//Do not write both AltiVec and scalar implementations in the same function!!!
void MyBrokenCode( … )
{
    …
    if( IsAltiVecAvailable() )
    {//Some vector code here }
    else
    {//Some scalar code here }
    …
}
```

The compiler inserts some AltiVec code at the beginning of any function that uses AltiVec, for the stack frame. If a non-G4 runs this code, it will hit it and likely crash. **Never turn off the VRSAVE instructions in the compiler preferences for MacOS apps, or you will get random errors.** What you have to do is write a separate vector and scalar version of the same function. This will make sure that the vrsave register will not be used until after you know it is safe.

```
//This way is better, but be careful if automatic inlining!
void MyNearlyFixedCode( … )
{
    …
    if( gIsAltiVecAvailable )
           DoItTheVectorWay();
    else
           DoItTheScalarWay();
    …
}
```

`MyNearlyFixedCode()` has one subtle weakness, however. It is possible that the compiler may decide to inline `DoItTheVectorWay()`, in which case its stack creation code will be moved to the beginning of `MyNearlyFixedCode()` and cause a crash. It is tedious to check every such function to make sure there was no inlining done automatically, and it would be a shame to have to turn it off. You may at your option turn off inlining on a per-function basis using pragmas around your function call. The following set of pragmas is for Codewarrior:

```
//This function is safer because the vector version wont become automatically
//inlined
void MyFixedCode( … )
{
    if( gIsAltiVecAvailable )
#pragma dont_inline on
           DoItTheVectorWay();
#pragma dont_inline reset
    else
           DoItTheScalarWay();
}
```

GCC has a function `__attribute__` that lets you do similar things. A more automatic (and therefore safer because you wont forget) method would be to use function pointers or C++ virtual class methods to prevent inlining. This has the advantage that you also no longer need the `gIsAltiVecAvailable` check each time the function is called:

```
//The C way: use a function pointer
void MyFixedCCode( … )
{
    (*DoItTheCorrectWay)();
}

//The C++ way: use a virtual base class to define an interface, and use
//a factory method to instantiate a child class of vector or scalar type.
void MyFixedCppCode( … )
{
    myObject.DoIt();
}
```

Special Topic: Optimization

When adding AltiVec to a pre-existing program, quite often you find that you need to vectorize a preexisting function. As an example, this is a function to calculate a third order polynomial of x:

```
// result = c0 + c1 * x + c2 * x^2 + c3 * x^3
float PolyNomial3( float c0, float c1, float c2, float c3, float x )
{
        return c0 + c1 * x + c2 * x * x + c3 * x * x * x;
}
```

I've prepared some sample code called "Polynomial" to accompany this section. You can play around with that if you like as you read along**. WARNING: the program will crash if it is run on anything earlier than a G4.** Before we continue, I should also explicitly mention that I have made no attempt to optimize the scalar version above beyond what the compiler already does. The morning after the release of the 1.0 version of this tutorial I was accused of constructing a paper tiger because the scalar version above did not contain the following optimization, called "Horner's scheme":

```
// result = c0 + c1 * x + c2 * x^2 + c3 * x^3 but faster
float PolyNomial3( float c0, float c1, float c2, float c3, float x )
{
        return c0 + x * (c1 + x * (c2 + c3 * x ));
}
```

…which is faster in principle because it compiles to three `fmadd`s rather than three `fmuls` and three `fmadd`s. In reality, I failed to make this optimization in both the scalar code and the vector code that you will see in a moment. It would benefit both. In the particular case of the precise scalar function shown above, this optimization doesn't make much of a difference because the extra `fmuls` do not add much to the cost of the scalar function due to pipelining. The pipeline is largely empty in the "optimized" Horner version because each `fmadd` depends on the result of the one before it. The three additional `fmuls` together only add two cycles to the overall execution time.

However, this optimization stands a good chance of accelerating the vectorized version where pipelining is addressed more aggressively. Since the pipelines are already full, reducing the instruction count by a factor of two means the AltiVec function could be twice as fast if the CPU is the performance bottleneck. Take this as an early lesson to make sure that you are using the right algorithm before investing heavily into AltiVec. [5]

[5] This is especially true, it appears, when your work is going to be open to public scrutiny! ☺

Retrofitting AltiVec into Scalar Code

Usually the first approach taken by most programmers when rewriting scalar code for AltiVec is to attempt to make the new AltiVec function fit into the mold of the old scalar version, using the same name and argument types and the same return value. This makes sense. The calling code won't have to change. So, let's do that for this function to see how well that works out:

```
//We will use this union type to move data from the FPU to vector unit
typedef union
{
        vector float    vec;
        float           elements[4];
}Float4;

//Our first attempt to vectorize PolyNomial3.
float PolyNomial3( float c0, float c1, float c2, float c3, float x )
{
   Float4  constants;
   Float4  the_Xs;
   float   returnVal;

   //Load some values into the vectors
   constants.elements[0] = c0;
   constants.elements[1] = c1;
   constants.elements[2] = c2;
   constants.elements[3] = c3;
   the_Xs.elements[0] = 1.0;
   the_Xs.elements[1] = x;
   the_Xs.elements[2] = x * x;
   the_Xs.elements[3] = x * x * x;

   //Now do constants · the_Xs  (Dot product)
   vector float result = vec_madd( constants.vec, the_Xs.vec, ZERO );
   result = vec_add( result, vec_sld( result, result, 8 ) );
   result = vec_add( result, vec_sld( result, result, 4 ) );

   //All the elements of result now contain the same value,
   //our result. Write it to returnVal so we can return it
   //as a floating point quantity
   vec_ste( &returnVal, 0, result );

    return returnVal;
}
```

Ok, lets benchmark this function and see how we did. Calling the floating point version 10,000 times takes 16733 time units. Calling our new vectorized version 10,000 times takes 46215 time units. Our AltiVec version is three times slower! Obviously we have done something wrong, but what could it be?

The problem with our approach is that the interface of the function itself is inherently scalar. This forces us to do so much data organization to set up the data for use by the vector unit that not only is the AltiVec speed advantage lost, we are actually three times slower than the simple FPU code. With a quick inspection, it should be apparent that almost all of it is stack overhead

— getting variables arranged where they need to be. Also, the `vec_add()` lines are doing a lot of redundant work, so even when we finally reach the stage that we are supposed to be operating efficiently in the vector unit, we aren't. There is no opportunity for pipelining here like there is in the scalar code.

The Fully Vectorized Approach

The solution is usually to redesign the function interface to be a vector interface and go back and tweak the caller a little. It isn't too hard, but it makes a huge speed difference! Here is a fully vectorized polynomial function:

```
// constants = { c0, c1, c2, c3 };
// x = four different x's that we evaluate at the same time
vector float Polynomial3( vector float constants, vector float x )
{
   //Expand out our constants and calculate x^2 and x^3
   vector float c0 = vec_splat( constants, 0 );
   vector float c1 = vec_splat( constants, 1 );
   vector float c2 = vec_splat( constants, 2 );
   vector float c3 = vec_splat( constants, 3 );
   vector float x2 = vec_madd( x, x, ZERO );
   vector float x3 = vec_madd( x, x2, ZERO );

   //result = c0 + c1*x[4]
   vector float result = vec_madd( c1, x, c0 );

   //result += c2 * x^2[4]
   result = vec_madd( c2, x2, result );

   //return result + c3 * x^3[4];
   return  vec_madd( c3, x3, result );
}
```

How did this new version do? On my machine, it evaluates 10,000 floats in 6410 time units. That is over twice as fast as the scalar code and nearly seven times faster than our first attempt at vectorizing this function!

So, what is the difference? First of all, we have completely gotten rid of all of the load/store instructions ...in this function, anyway. That was a huge overhead. Second, there is no redundant work here. Every element of every vector is serving a purpose. Furthermore, in roughly the same number of instructions or fewer as our previous example, we are evaluating the polynomial for four different X's at the same time! Finally, the code itself is straightforward, matching to a high degree the standard FPU code, making it *much* easier to read and debug.

Also, notice the difference between how we handled the data in this version compared to the last one. In the last version, each element in the X vector stood for something different: {1.0, x, x^2, x^3}. In this version, each element in every vector stands for the same thing as the other elements in that vector. That means that all the elements of the vector can be handled in the same way, which

is exactly what we want for a SIMD architecture. In our earlier approach, because our vectors did not contain similar elements, we ended up spending a lot of time shifting elements around maneuver them into the right place. Calculating in parallel works fastest when the 4, 8 or 16 streams are independent of each other.

However, don't mistake these results to indicate there is a hard and fast rule about how to handle data. There are a number of times when you don't have to parallelize your data in this fashion. For some tasks (e.g. inverting a matrix), where there is quite a bit of symmetry built into the operations, you can get reasonably good performance without having to invert four matrices at a time in parallel.

Adding Pipelining

OK, how do we improve this further? Well, we still need to work on scheduling. The `vec_madd()` function takes either 4 or 5 cycles to execute. We have three of them in a row, each of which depends on the result of the last one. For this reason, the three take 12-15 cycles to finish, instead of 6-7. The pipeline is hardly full. We are only completing one AltiVec instruction every 4-5 cycles, when we could be finishing one `vec_madd` per cycle, and also in principle make use of the vector permute unit at the same time. We can fill the VFPU pipeline by evaluating four vectors in parallel. This is essentially done by "unrolling the loop", a common trick for writing blitters. Once again, we have had to go back and edit the caller, this time to make it pass us a pointer to all of the data, instead of small bits of it at a time. Loop unrolling makes for very large code, so I won't show it here, but it is in the Polynomial program for you to look at.

Another approach I could have taken instead is to declare the function inline and hope that the compiler was able to schedule the instructions in with whatever the caller is doing. This can be particularly beneficial because AltiVec stack overhead tends to be large. When an inline function is called within the confines of a loop, the compiler may automatically unroll the loop allowing the function to be interleaved with itself many times over, achieving the same effect that I worked hard to produce by hand. Unfortunately, the compiler can be picky about what to inline, so it doesn't always work. The function generally must be small. In addition, for the purposes of a tutorial I wanted to show the explicit unrolling of the loop so you get to see what it looks like. How well does evaluating four vectors in parallel enhance performance? We can now evaluate 10,000 polynomials in 2400 time units! That is over seven times as fast as the scalar code.

Optimizing Cache Usage

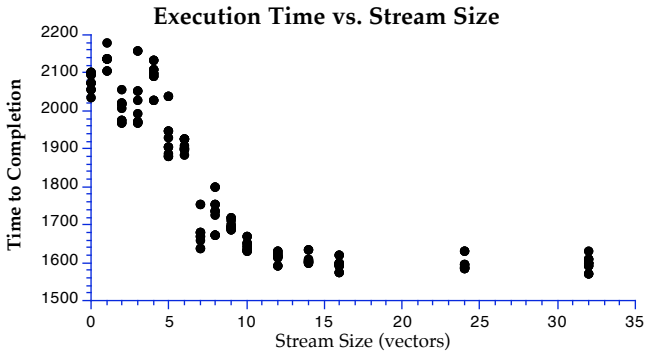
Are we done yet? Well no. To see why, we will examime our routine with **Sim_G4** (See the section entitled "Sim_G4" for a description of this invaluable utility.) Let's take a look at how well we are executing so far. The below trace shows the actual amount of time each instruction takes inside our function's main loop. (This is just a snapshot of one particular pass through the loop.) Each instruction is listed on the left, then the clock at which the instruction started, a graphical display of what it was doing each tick, and finally a number showing during which clock the instruction finished. Each instruction goes through four or five stages. It is fetched in the instruction buffer (**I**), Dispatched (**D**) to the appropriate execution unit, executes (**E**), and retires (**R**). If there is an instruction ahead of it in the completion buffer, then it will display (**F**) for a number of ticks until the item ahead of it in the completion buffer is retired. Two instructions can be retired per cycle on 7400/7410, three on 7450:

```
2136:addi    R9,R0,0x0    |  7116 | .....IIIIDFFFFFFFR..................................................................... | 7128
2137:addi    R8,R0,0x10   |  7116 | .....IIIIIDFFFFFFFR..................................................................... | 7129
2138:lvx       V0,R9,R3   |  7118 | .......IIIIDEEE
```

memory unit was asked to provide data and the data was neither in the L1 or L2 cache, so it had to take a long, slow trip to main RAM for it.

The solution is to add in cache instructions to help the CPU anticipate what data it is going to need. In the fourth example function in the Polynomial app, I've added a call to `vec_dstt()` to make sure our source buffer is loaded in time. In earlier versions, I also called `dcbz` to zero the blocks that we are writing to before we write to them, however I have since removed it. Why do that? When you issue two back to back vector store instructions to the G4 and the two vectors belong to the same cacheline, the issues a kill transaction that allows you to write to a cacheline without loading it, much like dcbz allows you to do. This is part of the store miss merge mechanism. This method is potentially faster than dcbz based cacheline prefetching on a 7400 because dcbz can only be issued every fourth cycle and occupies extra space in the instruction queue, potentially displacing something else that could have been usefully done. On a 7450, the situation is a bit less clear because

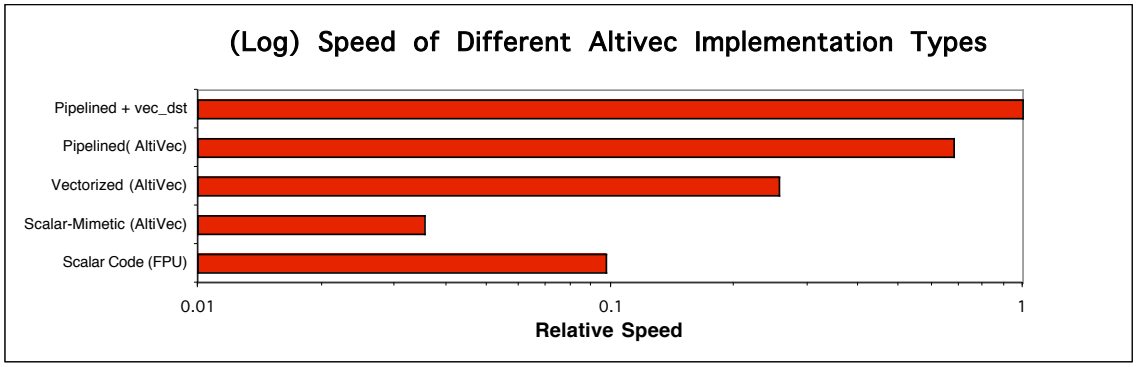
Using `vec_dst` or `vec_dstt` also requires some care. While you could attempt to stream in the entire input data set at once, in practice this generally doesn't work very well because interrupt level code or other preemptive threads may interrupt and call `vec_dst` on the same stream, halting your stream and replacing it with its own. Also the stream may outpace your code, displacing needed data with data we don't need yet.



Typically what you want to do is set up many small overlapping streams. In each loop iteration, ask for a small stream that reads 64, 128 or 256 bytes forward from your current location in memory. It is ok to repeatedly use the same stream id. Try to stay away from id 3. `BlockMoveData()` uses it frequently at interrupt level. How many bytes to read ahead usually must be determined experimentally. Generally there is a number beyond which no performance advantage is seen. If your data set sizes vary, you may also need to check different data sizes. In this particular case, the optimum stream size was in the 10–16 vector range (5-8 cache blocks). Notice that I gathered a lot of data. There is some fluctuation in the numbers that you get, so usually you have to sample each data point a few times. I repeated each five times.

The combination of cache streaming and zeroing cache blocks improves the efficiency of our memory use a bit. We can now do our task in 1600 time units — ten times faster than the FPU

and nearly thirty times faster than our first vector attempt! Here is a graphical representation of our different implementations. Longer bars are better. Note that this is Log_{10} scale:



Quick inspection of these results reveals that although we did see a 25% rate acceleration due to cache hints, we did not apparently get back all of those 50 cycles wasted per loop. If we had, the speed might have more than doubled. Unfortunately, at this point we have probably run into a fundamental weakness of the hardware. The memory subsystems are woefully inadequate to keeping the vector unit properly fed when running at a full gallop.

Sim_G4 reports that adding cache hints does drastically accelerate the function for about four or five loop iterations. However, after that point, we start to stall again in lvx (though in dispatch, rather than execution). What appears to be going on here is that the first time through the loop, the code is running very slowly because the instructions themselves are being loaded in from RAM. This gives the memory unit plenty of time to pre-fetch some data. However, in the second or later iterations through the loop, the instructions are already loaded, so we are able to proceed at maximum speed. We quickly catch up to the data stream and then start to stall again. Fortunately, because we are still prefetching data, the stalling isn't quite as bad as it could be, but it is quite significant. Often one stall will delay long enough that the next cacheline loaded from won't miss, so we only stall some of the time.

The only thing that we can do now to get more speed is to do more with each piece of data before we store it. It looks as though our calculation could easily be 3 times as complex and still run at memory fill rates. If we had something else we wanted to do with this polynomial, such as calculate where the points go when plot it out on screen, we could probably do that now and get the extra math for free. Sadly that is beyond the scope of this tutorial. It is something you will have to experiment with in your own program.

What about Horner's optimization discussed at the beginning of this chapter? I was curious so I gave it a try. It accelerated the scalar code by 8% and the AltiVec code by 3%. That is not quite the factor of two that was claimed based on just counting instructions! Actually, once you take execution

times and the fact that some of the fmuls can pipeline into account, it should be 9 cycles for Horner and 11 for the original scalar version on 7400, so we really should only predict a 22% acceleration based on the instructions themselves. We don't see much speed improvement for the AltiVec code either, but since we already know we are limited by the speed of memory, it isn't too unexpected. Clearly, it isn't just what you code, it is how you call the code, when you call it, and where the data is! In this case, we didn't improve any of those other things and so after a point just improving the implementation of the function itself didn't do too much good. You can do yourself a lot of good by paying attention to all facets of how your program is constructed, including how data is passed into a function, how data is stored in memory, pipelining, temporal locality, your use of constants, etc.

Summary

Hopefully by now, you have seen that the AltiVec optimization process is much like for other code, perhaps with a few extras thrown in:

(1) Only optimize those functions that are frequently called and are the performance bottleneck in your application. A good profiler is a must.

(2) Find the best algorithm. While AltiVec might buy you a factor of ten in performance, it surely isn't going to get you a factor of one hundred or one thousand. Often you can get that by doing something a different way. Picking the best algorithm also benefits your scalar version. You can still accelerate that with AltiVec.

(3) Once you have found the best method, arrange it for maximum parallelism. If you find you are doing a lot of permute operations to shift vector elements around relative to one another, it is a bad sign. The best implementations tend to have vectors in which all elements in the vector can stay in place and be processed in parallel. You may have to go back to rewrite the caller a little bit to make sure that the data is handed to you in a useful format. Look to find ways to reduce memory overhead, either by passing constants and globals in as arguments or by generating them on the fly, don't waste too much time creating constants. At worst you can load in a cacheline full of constants, if you need to.

(4) Optimize your function for optimal instruction scheduling. If you use the VFPU or VCIU, typically this means that you will be processing data in a loop 64 bytes at a time so that you can have 4 independent vectors to stuff the pipelines with. If your function takes its data passed by value, either declare the function inline or take multiple vectors full of data at once. If your function reads data from memory, unroll your loop a little to read four or more vectors at a time. Do not unroll the loop completely because this will mean more instructions will have to be loaded into the cache, which may hurt performance.

(5) Only once you have done all other optimizations should you start looking at cache instructions. This way your memory access patterns are set in stone. If your function does any memory access, quite often it looks a bit like a blitter at this point:

```
//AltiVec functions often start to look like blitters.
//Here is the suggested placement for cache instructions.
void DoSomething(vector char *inputData, vector char *outputData,int dataSize )
{
    //Start the first prefetch as early as possible.
    //The best block size depends on the function –
    //typically it has to be determined experimentally.
    int     prefetchConst = 0x10010100;     //256 bytes
    vec_dst( inputData, prefetchConst, 0 );

    vector char v1, v2, v3, v4;

    int loopCount = dataSize / 64;
    while( loopCount--)
    {
           //Prefetch at the start of each loop
           vec_dst( inputData, prefetchConst, 0 );
           v1 = inputData[0];
           v2 = inputData[1];
           v3 = inputData[2];
           v4 = inputData[3];

           ...

           outputData[0] = v1;
           outputData[1] = v2;
           outputData[2] = v3;
           outputData[3] = v4;
           inputData += 4;
           outputData += 4;
       }

    //Stop the stream as soon as you no longer need it
    vec_dss(0);
}
```

Calculate your prefetch constant and place a call to `vec_dst()` at the very beginning of your function. This ensures that while you are going through the relatively slow process of loading in the instructions for the function you can also be prefetching the data that you need. Also place a call to `vec_dst()` at the start of the loop and call `vec_dss()` for the stream at the end of the function.

There is no one correct stream block size that fits all functions. Typically, you have to test experimentally to find out what the best size is going to be. This can be done in the context of a test app. Make sure that your data set resembles a real data set if it is likely to impact performance. Take multiple data points for each block size as the times can be somewhat variable. Typically block sizes in the range 64-256 bytes work best. For 2D buffers, a successful strategy is often to fetch the next row while you are working on the current one.

Do not use __dcbz() to zero destination blocks that you intend to overwrite. The reason is that the G4 supports store miss merging that will merge stores to adjacent vectors on the same cacheline to a single cacheline write. On most machines this feature removes the need for dcbz because it automatically issues a kill transaction like dcbz but may have better performance characteristics than dcbz. Note that for this to work, it is important that the two vector stores occur back to back or as close to each other as possible in the instruction stream.

A good rule of thumb is that unless you are eating up at least 15-20 cycles of CPU time per vector load (after pipelining) and your data has to be loaded in, you are probably memory rate limited. This means that you will be stalling on loads and backing up the completion queue. If you can find more work to do per vector this can greatly accelerate your application. You will not only get more done per load/store pair, you will also be able to do memory access in parallel with data processing. Code running at this level of complexity will run at the speed of the CPU rather than the memory bus, a very desirable thing. If you can achieve this level of complexity in your function, it no longer matters whether your data starts in RAM or the caches. For this reason, this is a very good situation to investigate the transient cache instructions and LRU loads and stores with your primary data stream. This will help leave data that depend on the caches for fast processing in the caches.

(6) Move the function into your app and see if vec_ldl(), vec_stl() or vec_dstt() work better or worse in place of vec_ld(), vec_st() and vec_dst(). Since the transient / LRU versions tend to speed up code around your function at the expense of the function itself, its performance impact is difficult to measure correctly in a test app where there are no surrounding functions to benefit.

If you have lots of time to waste, go back and repeat steps 5 and 6 to see if a different block size works better with the new cache instruction variants you added in step 6. Also check performance on different machines.

Other Resources

www.simdtech.org and altivec@simdtech.org

The simdtech.org site houses some articles on using AltiVec for different tasks, and also its use on other platforms like Linux. It is the same thing as the old AltiVec.org site, but it has been expanded to include the new e500 vector architecture for embedded applications.

http://www.simdtech.org/

Most signficantly, the **altivec@simdtech.org** mailing list to be found there is the best place to field difficult *AltiVec* questions. Hopefully once you have read this tutorial, you won't have to post there asking, "How do I get started?"

http://www.simdtech.org/altivec

Just in case, there is a parallel yahoo list mirror that stores the archives of the mailing list since Spring 2001. It is searchable. In addition, there are links to *AltiVec* hardware and software providers and other handy stuff there. You can subscribe to the group to get the forum@altivec.org list in a digest format

http://groups.yahoo.com/group/altivec

Apple Sample Code

Apple has posted some sample code that uses AltiVec, that does Fast Fourier Transforms, Multiprecision Math, and Wavelet stuff. These can be downloaded here:

http://developer.apple.com/samplecode/Sample_Code/Devices_and_Hardware/Velocity_Engine.htm

http://developer.apple.com/hardware/ve/download_summary.html

Apple's Velocity Engine (a.k.a. AltiVec) Pages

Apple has some extensive pages describing the performance benefits of using the vector unit:

http://developer.apple.com/hardware/ve/

There is a particularly nice list of AltiVec related downloadables here, including **BLAS**, a **vectorized MathLib**, **matrix operations**, and even a version of **MacsBug** that has AltiVec **dcmds**:

http://developer.apple.com/hardware/ve/download_summary.html

This site has been extensively overhauled, with about double the amount of new material there, so if you haven't visited in a while, it is worth a look. You can also find some competitive AltiVec benchmarking analysis against similar software on other processors.

http://developer.apple.com/hardware/ve/summary.html

Apple's Performance Tools and Inside MacOS X: Performance

Apple has a suite of other performance tools available. Most of them are probably already on your computer, if you are using MacOS X. You can find extensive documentation in "Inside

MacOS X: Performance" on their use. In addition, that volume provides a lot of OS level wisdom about how to organize and use memory efficiently in your application:

http://developer.apple.com/techpubs/macosx/Essentials/Performance/performance.pdf

Apple Hardware Architecture Performance Group's CHUD Toolkit

CHUD is a framework and collection of apps for detailed performance analysis of your code. It contains a powerful sampling application called Shikari (similar to VTune if you have used that), MONster for using multiple performance monitors concurrently, and Amber and Sim G4 for cycle accurate simulation of code performance (or lack thereof) on PowerPC 7400:

http://developer.apple.com/tools/debuggers.html

Motorola

Here are the manuals for the PowerPC 7400, 7410, 7450 and 7455. (Also called MPC7400, MPC7410, MPC7450, and MPC7455):

http://e-www.motorola.com/brdata/PDFDB/docs/MPC7400UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7410UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7400UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7410UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7410UMAD.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/MPC7450UMAD.pdf

Motorola also published a guide to help explain the differences between these processors.

http://e-www.motorola.com/brdata/PDFDB/docs/AN2203.pdf

Here again are links to the *AltiVec™ Technology Programming Interface Manual* (PIM) and the *AltiVec™ Technology Programming Environments Manual* (PEM), the essential references for AltiVec:

http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf
http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPEM.pdf

The AltiVec Web Ring

There is a web ring linking a series of AltiVec sites together. These will provide you with helpful advice, tips, tricks, and even (if you dare) an AltiVec humor page!

http://home.san.rr.com/altivec/index.html

The IBM PowerPC Compiler Writers Guide

This handy tricks reference manual is full of examples of code fragments that perform really well on PowerPC for various purposes.

http://www.chips.ibm.com/products/powerpc/tools/compiler/cwg.pdf

People who like this sort of stuff may also enjoy Hacker's Delight (ISBN: 0201914654):

http://search.barnesandnoble.com/booksearch/isbnInquiry.asp?userid=55B1LAXWEV&isbn=0201914654

Holger Bettag's Vector Constants Page

If you need a quick way to prepare a vector unsigned char constant for use in a function in 1-4 operations, Holger's page might be just the thing.

http://www.informatik.uni-bremen.de/~hobold/AltiVec.html

He is working on doing the same thing for 16 bit constants or perhaps even 128 bit constants.

The Alienorb *AltiVec* Page

This is my site with further information, tips and techniques for writing *AltiVec* code. There you will find more programming examples, handy stuff that didn't quite fit this document, workarounds for *AltiVec* limitations, and how to do some things that you probably ought not to be doing (suitable warnings are provided), and of course links to still more sites. It is a bit easier to edit than this document, so new stuff goes there first:

http://www.alienorb.com/AltiVec

If you have any suggestions for additional material be sure to **contact me.**

> The alienorb.com website has been down for about a year. Expect a triumphant return in the near future, courtesy of idevgames.com.

TroubleShooting

Q: At random times on MacOS, my vector registers are inexplicably overwritten by 0x7fffdead. This is especially common when virtual memory is on or on MacOS X.

A: This happens when some other process or interrupt level task momentarily stops your thread and begins executing. Normally, the VRSAVE special purpose register is supposed

to be configured to tell the new thread which registers to save before it switches contexts. If you have not turned on the "Generate VRSAVE instructions" switch in the project preferences, then the VRSAVE register may not be set up properly. The result is that every time you are swapped out, those registers that correspond to bits in the VRSAVE register that are set to 0 will have their contents replaced by 0x7FFFDEAD (□ 4). This is especially common when virtual memory is on because paging will have this effect too. **Usage of the VRSAVE register is required on MacOS**. On other OS's it is typically ignored or set to 0xFFFFFFFF.

If turning on the "Generate VRSAVE instructions" switch in the project preferences does not solve the problem, you may have encountered a bug in CodeWarrior 6.0 or 6.1, which will sometimes fail to correctly set the VRSAVE register in functions that use only vector load/store or vector compare instructions. The only way to solve that problem is to update your compiler, or use a different compiler.

Q: Damn, this is hard!

A: Most functions that turn out to be miserably hard to write are hard to write because your data is poorly organized for parallel processing. Try to have each vector contain only one type of data. If your vector contains an X, a Y and a Z member, try reorganizing that into a vector full of X's, a vector full of Y's and a vector full of Z's. Also keep your data together so that it can be loaded and stored with a single vector load or store. It's a lot easier that way!

The problem seems to stem from data organization habits for scalar code, which is to say no organization discipline whatsoever.

Q: I seem to be getting incorrect answers but my calculation ***IS*** right. What happened?

A: Assuming you aren't falling victim to the VRSAVE problem (**see above**), then there are two additional common problems to check for:

The first is data alignment. Remember that the address of vector that you load will be rounded down to the nearest multiple of 16 before the load occurs. The same goes for vector stores. Even if your code handles alignment correctly, you can still get alignment bugs due to bugs in other people's code:

> CodeWarrior 5.3 returns misaligned arrays when array new is used. They are 8 byte aligned but not 16 byte aligned.
>
> Your debug code may prepend a header onto heap blocks that is not 16 byte aligned.

> Certain versions of MPW/MrC don't preserve vector alignment in classes / objects when they appear as a sub-class or member datum of another class.
>
> Finally, GCC may fail to correctly align structs or unions containing vectors if the function they appear in is inlined into another function.

If all of your results are wrong or in the wrong place, this is commonly the problem.

If on the other hand, only *some* of your results are wrong, then the other problem you can run into is a bug in the OS that does not correctly restore the vector registers when a OS function is called. For example, MacOS 9.2.1, 9.1 and perhaps earlier versions sometimes zero vector registers 20 through 31 and VRSAVE when `WaitNextEvent()` or `EventAvail()` are called. Since `printf()` calls these in MSL, `printf()` may have this effect too.

> Note: this will make debugging by `printf()` a lot harder! There are other error logging utilities such as **DCon** available that may be used instead. In addition, you may simply use `Debugger()` or `DebugStr()`.

Finally, be aware that correct handing of unaligned vector stores means that you read and then write back unchanged the extra 16 bytes of data surrounding an unaligned vector to be written. If preemptively multithreaded code (or interrupt level code) changes those surrounding bytes while the alignment calculation is ongoing, the change will be overwritten. It is not a good idea to store any data being changed by other threads or interrupt tasks (also mutexes, OTLocks, MPSemaphores, signal bits, etc.) near unaligned vector data for this reason.

Q: My code has a LOT of seemingly redundant load and store instructions. Performance is terrible. How do I get rid of them?

A: This usually happens either because the optimizer is turned off in the compiler, or (in certain versions of GCC 2.9.x) because of a register coloring bug.

Q: My code spends a lot of time pointer chasing. How do I optimize this for AltiVec?

A: Remove the pointer chasing. The load store unit can only load and store one quantity per cycle. There is no such thing as a vectorized load or store (scatter loads and scatter stores where data is loaded into a vector from multiple separate memory locations). There is just loading and storing of 128 bit contiguous vectors. Find some way to store your data in contiguous arrays so that you dont have to deal with so much memory indirection.

Q: I can't figure out how to efficiently mix object oriented designs and SIMD. How is that done?

A: Often people looking to tackle this problem are hoping to get all the flexibility of C++ and the speed of AltiVec. Instead they all too often end up with the speed of C++ and the flexibility of AltiVec! If you are in this position, you have my sympathy.

As this is the subject for another article that I plan to get around to writing eventually, I don't plan to offer too many spoilers. However, I will say that the key to solving this problem is to recognize that objects are frequently data collections of non-identical type and vectors are (or should be) collections of identical type. Furthermore, C++ itself frequently employs lots of light-weight objects, while AltiVec is not very good at being a low latency engine.

Does that mean that SIMD and OOP are orthoganal? Not at all! It does however mean that you have to thoroughly understand the strengths of each to design object architectures that make sense for Altivec. Otherwise, they can and often do work across purposes. I plan to present some design motifs that work well for both, when and if I do manage to get it out the door.

Acknowledgements

A number of fine programmers have contributed to this work. Some of the optimized scalar code should be credited to Holger Bettag. Holger has also served as a mentor to many of us on the AltiVec mailing list, educating us as to hardware specs for current and forthcoming processors. Alex Rosenberg's knowledge of MacOS arcana is unparalleled, and contributed greatly to this work. A big thank you also goes to Holger Bettag, Jean-Manuel Prudon, Sandra Nielsen, David Duncan, Craig Mattox, Carl Manaster, Chuck Fleming, Anton Rang, Steve Poole and Sam Vaughan for their invaluable suggestions and work spotting errors in the text. Steve Poole has helped me port some of the sample code to linux. Finally, many thanks to the Apple Hardware Architecture Performance Group for CHUD and for answering a great many questions, and to Craig Lund (Mercury) for supporting the AltiVec mailing list all these years.

Contacting the Author

If you have questions, comments or corrections for this tutorial, please do not hesitate to send them to me, Ian Ollmann: mailto://iano@cco.caltech.edu